FreeBSD® JOURNAL

January/February/March 2025

Topic: Downstreams

FreeBSD Release Engineering: A New Sheriff is in Town

GhostBSD: From Usability to Struggle to Renewal

BSD Now and Then

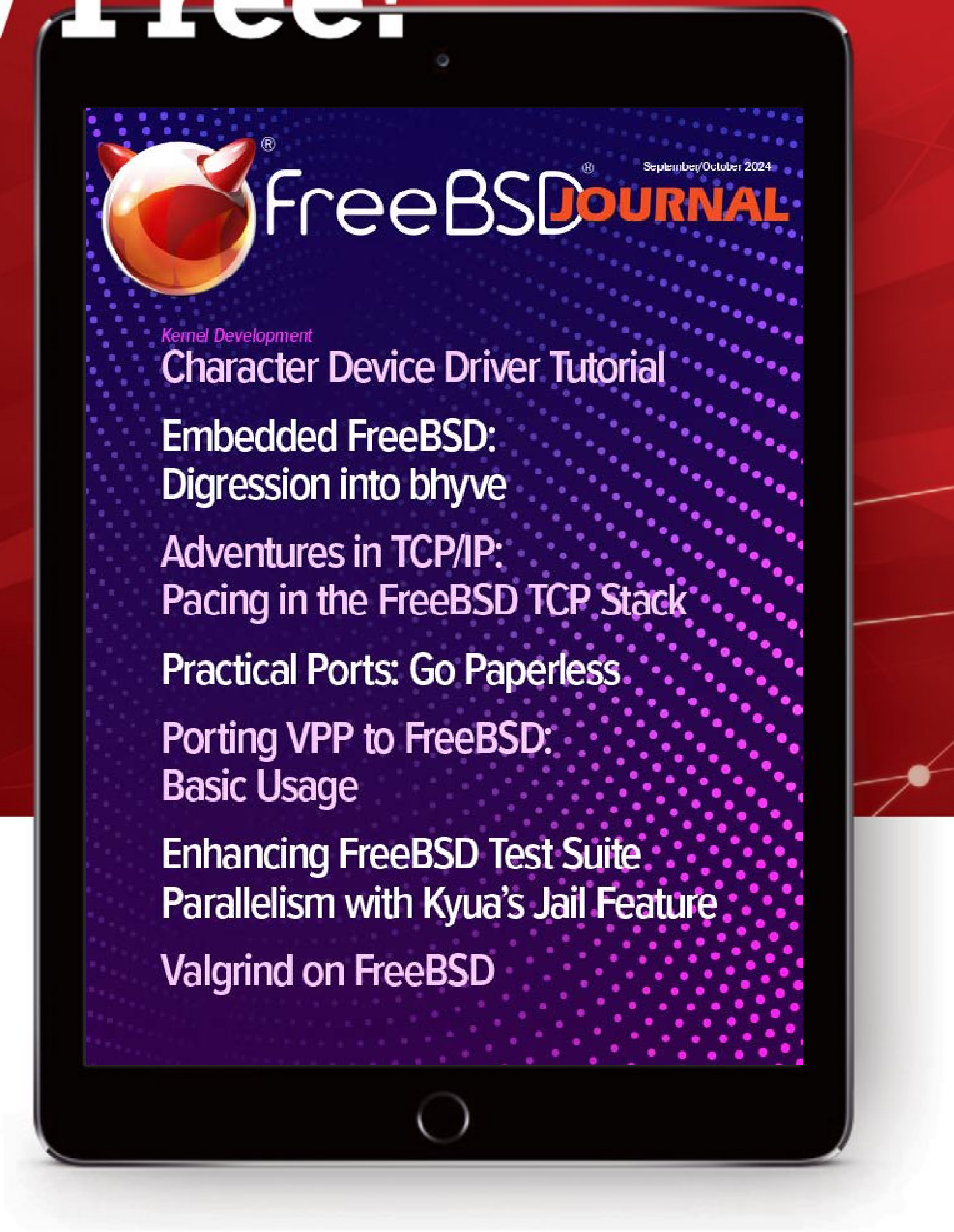
Character Device Driver Tutorial (Part 3)

The FreeBSD Journal is Now Free!

Yep, that's right Free.

The voice of the FreeBSD Community and the BEST way to keep up with the latest releases and new developments in FreeBSD is now openly available to everyone.

DON'T MISS A SINGLE ISSUE!


FreeBSD Journal
September/October 2024
Kernel Development
Character Device Driver Tutorial
Embedded FreeBSD: Digression into bhyve
Adventures in TCP/IP: Pacing in the FreeBSD TCP Stack
Practical Ports: Go Paperless
Porting VPP to FreeBSD: Basic Usage
Enhancing FreeBSD Test Suite Parallelism with Kyua's Jail Feature
Valgrind on FreeBSD

2025 Editorial Calendar

- Jan/Feb/March Downstreams
- April/May/June Networking
- July/August/Sept Contributing/Workflow
- Oct/Nov/Dec Embedded

Find out more at: freebsdfoundation/journal

S&W PUBLISHING LLC
PO BOX 3757 CHAPEL HILL, NC 27515-3757

Editor-at-Large • James Maurer
maurer.jim@gmail.com

Design & Production • Reuter & Associates

FreeBSD Journal (ISBN: 978-0-61 5-88479-0) is published 6 times a year (January/February, March/April, May/June, July/August, September/October, November/December).
Published by the FreeBSD Foundation,
3980 Broadway St. STE #103-107, Boulder, CO 80304
ph: 720/207-51 42 • fax: 720/222-2350
email: info@freebsdfoundation.org

LETTER from the Foundation

Welcome to the first *FreeBSD Journal* issue of 2025! This is the Journal's first quarterly issue following our new schedule announced in the fall of 2024 and it packs a wide range of articles, a couple of conference reports, and yet another treat from Michael's witty pen. Speaking of conferences, registration has just opened for BSDCan 2025 where several of the editorial board members will be present. Come and join us as we are always happy to chat about FreeBSD.

BSDCan 2025 will also be the site of the annual FreeBSD developer summit. This year's summit will be bustling with activity as the developers nail down the final plans for 15.0 which is slated for release at the end of this year. Colin's article covers the upcoming schedule for 15.0 as well as other future releases.

Eric Turgeon introduces the GhostBSD project including an overview of its history to date and future plans. Michael and Randall continue their Adventures in TCP column with a deep dive into SYN segments. Christopher Bowman's latest installment uses FreeBSD's GPIO subsystem to control LEDs. I wrap up my three-part series on character device drivers with an article covering memory mappings. Finally, the cast of BSD Now provides a retrospective on 600 episodes of their popular podcast.

As always, we love to hear from readers. If you have feedback on any of our articles, suggestions for a future article, or are interested in writing an article, please email us at info@freebsdjournal.com.

John Baldwin
Chair, *FreeBSD Journal* Editorial Board

January/February/March 2025

FreeBSD® JOURNAL®

We Get letters

by Michael W Lucas

Mister Letters Answerer,

Lots of people have built stuff on top of FreeBSD, bundled it up, and made it something you can install. I have an idea for something like that. What do you think?

—Gonna Bring My Ideas to Life

Dear GBMItL,

Interesting idea. I bet you're still young enough to remember hope.

When I built my first Unix desktop running FreeBSD 2.0 whatever I thought I would master this operating system. It had source code, and I could read C! I sincerely believed that a meticulous line-by-line assessment of the operating system's innards would grant me a skill level unknown to anyone except those anonymous faceless gurus generously volunteering their services on the freebsd-questions mailing list.

Like you, I had not yet learned to meticulously assess randomly appearing but tasty-smelling enticements for treachery.

I've written about my exploits with the FTP source code elsewhere in this column and see no need to taint your soul with a second retelling, even though extensive therapy and microdosing thrift store Shirley McClain "reveal your past life" audio cassettes have exposed further details of that odyssey. You learn that I once used a script to identify which FTP code called which function, order the source files as such, and send the result to my employer's industrial-scale printer only to discover several thousand pages later that the code was recursive would serve nicely to dispel any illusion you might suffer from that I possess any expertise whatsoever, but it would not further illuminate you on the subject of recursion or downstreams or, indeed, hope. It would suggest that experience is what you get when your employer makes you buy a new toner cartridge.

Like you, I had not yet learned to meticulously assess randomly appearing but tasty-smelling enticements for treachery.

It also did me the favor of ablating another layer off my already-stubby hope.

I've also mentioned FWTK in this very column—my fourth, if I recall correctly. Let me check my copy of available-at-sketchy-and-dubious-retail-outlets-near-you *Dear Abyss: The*

FreeBSD Journal Letters Column, Years 1-6 to verify—yes, the fourth. But I didn't talk much about it.

FWTK. The Fire Wall Tool Kit. Yes, Fire Wall. In the 1990s we had not yet invented compound nouns. Firewalls cost tens of thousands of dollars, but FWTK let you install proxies for HTTP and SMTP and more on a cheap Unix box with two network cards. All connections from inside your organization terminated on the FWTK box, which would go fetch the requested resources and hand them back to you. It supported all the modern protocols, like HTTP 1.0 and Gopher, and, through the plug interface, your ISP's Usenet server. What else did you need? It did the job reliably.

The license forbade commercial use. You couldn't resell FWTK.

But you could independently consult to install and maintain FWTK.

I was an independent consultant, wandering from company to company with my laptop and belled hat, fixing networks by application of undying principles like "use a crossover cable to connect hubs" and "PCs choked by airborne filaments fiberglass insulation will overheat and catch fire because, you know, *insulation*." The age of script kiddies had just begun and a whole bunch of local legal firms wanted one of those fancy firewall thingies. Legal firms seemed like great clients: someone who charges $250/hour doesn't flinch when a consultant charges half that. You're only claiming to be half as worthwhile as they are, after all.

Firewalls cost tens of thousands of dollars, but FWTK let you install proxies for HTTP and SMTP and more on a cheap Unix box with two network cards.

Installing FWTK was not hard. You downloaded the source code. You compiled it. You stuck the binaries in /usr/local/bin. A couple of config files later and boom—you had a firewall comparable to the big expensive ones, and best of all, the customer's money went into *your* pocket.

FreeBSD's FWTK port made the process a doddle, but was I satisfied? No, I was not. Because, you see, I had the source code. Source code makes you ambitious. Source code makes you think you can do anything. The world is stuffed with people who discovered they could read the source code and suddenly thought they could do anything, like innovate Internet payments, reinvent automobile drivetrains, build actual 1950s-style rocket ships, or reconstruct government without understanding anything about any of these. They sometimes make fortunes on the way but inevitably fail in vituperious disgrace and vitriolated shame, because being smart enough to read source code has absolutely nothing to do with competence or being a worthy human being.

I thought I could make things better. Ah, hope, such sweet toxicity!

If I could build my own FreeBSD that already included FWTK and the appropriate configuration files, perhaps even with those files in pre-initialized RCS version control, I could cut the installation time down from four hours to one. Just think of what I could do with those extra three hours! I could… search desperately for another client who needed an FWTK install, that's what I could do. Never mind that the entire point of hiring a pricey consultant is that the client gets to see them sweating for their benefit.

All I had to do was slam the source code into /usr/src/contrib, edit a Makefile and boom—when I built the operating system it would build the port. (We also didn't have freebsd-update(8) in those days. Every security patch meant building the affected components from source. FreeBSD didn't have the Open Group UNIX™ certification, but it was unquestionably chest-thumping Real Unix.

Like everything involved in the hideous nerd sport of Computer Touching, it failed.

I copied the error messages into Stack Exchange—no, wait, we didn't have such websites then. The mighty search engines of the 1990s were primarily designed for querying by fetish. The FreeBSD website had a search engine, however, and the mailing list archives were indexed. I learned, I fixed my errors, and I created new errors. Not exciting errors. Or useful errors. Just errors, to be churned through and overcome and used to create more errors.

I spent weeks of my free time on that project, where "free" was defined as "stolen from family, doing dishes, and bathing."

But once I integrated it, I would be able to run my own release, burn it to CD, and have a fully patched, installable firewall. Never mind that FreeBSD didn't yet have the release(7) man page. The mailing list archives had notes on how it was done!

I spent weeks of my free time on that project, where "free" was defined as "stolen from family, doing dishes, and bathing."

Keep in mind that I was only trying to integrate contributed software. I wasn't doing anything like, say, those HardenedBSD maniacs trying to change core kernel code while simultaneously maintaining synchronization with FreeBSD itself. On the other hand, HardenedBSD has the "advantage" of using git rather than CVS. (Making people believe that software forks are sensible, maintainable, or sustainable might be git's greatest crime, but anyway.)

I kept at it even after I ran out of law firms that needed a firewall. I wasn't going to let a stupid chunk of computer code defeat me. After all, if a guy like *the* Jordan Hubbard—who had notoriously read the wall(1) man page, thought "It can't work like that," and promptly sent a message to every single Internet user in the entire world—could do it, I surely could!

I failed.

Here's the difference between Jordan and me. He's willing, even eager, to fail at scale. In front of literally everyone.

Jordan, Rod Grimes, Nate Williams, Mike Smith, and all those folks put their work into the world and attracted other people to their vision. They were smart enough to know that the source code didn't grant them phenomenal cosmic powers but instead kept at it out of pigheadedness and the will to create something cool and useful, unlike my pigheadedness and greed.

Occasionally, they even documented that vision.

And here we are.

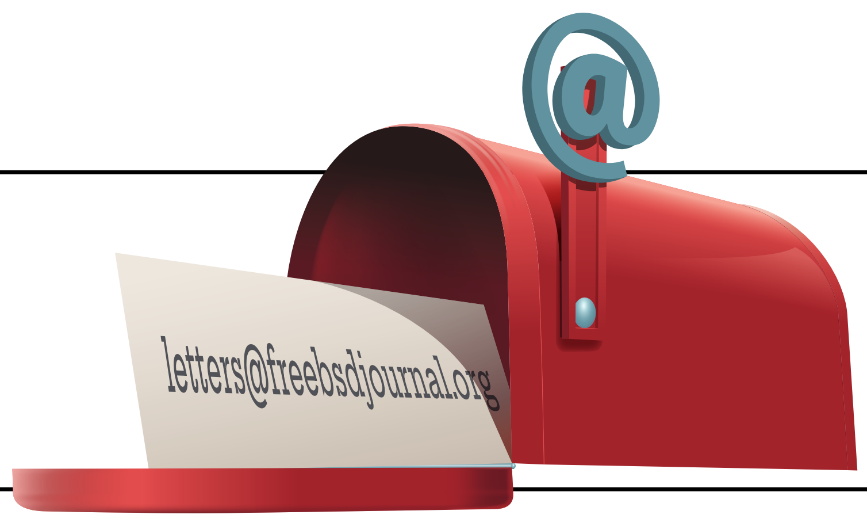
So, should you make a downstream? Do you have a vision? Do you have an infinite capacity for overcoming your own mistakes? Are you willing to tell the whole world what you have done? Then go volunteer for a worthy cause helping those less fortunate than you, because that's how you make changes in the world!

Yes, that involves leaving your keyboard. Sorry.

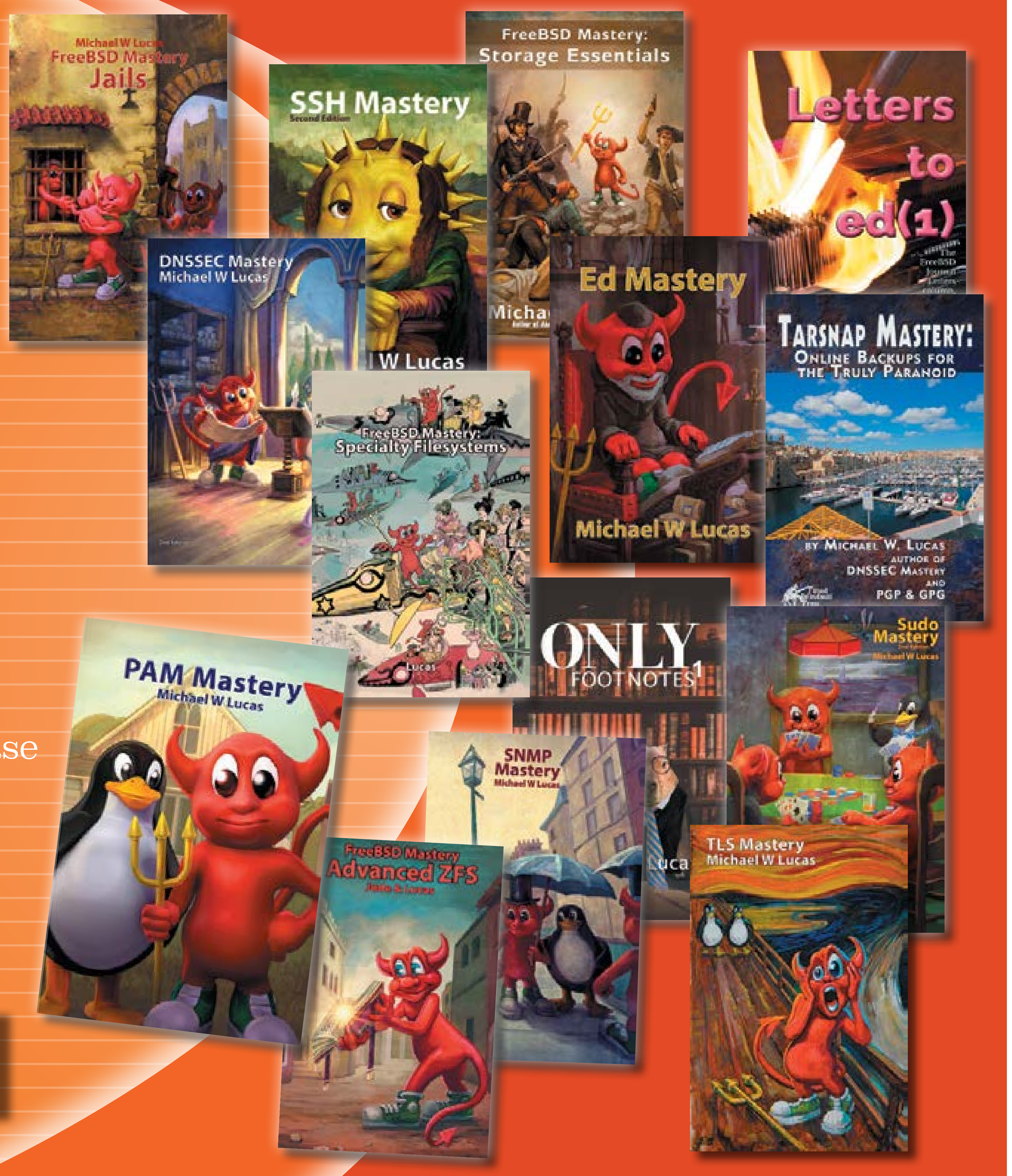
Fine. Build a downstream. See if I care. Don't repeat my mistakes, though. Go make your own. It's the best way to learn!

Also: whenever a legal firm invites you to work for them, check with other contractors they've used first. Fighting a law firm over unpaid invoices is difficult. After all, they have all the lawyers.

Have a question for Michael?
Send it to letters@freebsdjournal.org



MICHAEL W LUCAS is the author of *Absolute FreeBSD*, *Run Your Own Mail Server*, and too many other books. He probably needs his medication adjusted. Again. Learn more at https://mwl.io.


FreeBSD Mastery Jails
SSH Mastery
FreeBSD Mastery: Storage Essentials
Letters to ed(1)
DNSSEC Mastery
Ed Mastery
Tarsnap Mastery: Online Backups for the Truly Paranoid
FreeBSD Mastery: Specialty Filesystems
Only Footnotes
PAM Mastery
Sudo Mastery
SNMP Mastery
FreeBSD Mastery Advanced ZFS
TLS Mastery
Michael W Lucas

Contents

FreeBSD Release Engineering: A New Sheriff is in Town

BY COLIN PERCIVAL

On November 17, 2023, Glen Barber retired from the position of FreeBSD Release Engineering Lead after a decade of managing FreeBSD releases, and with the endorsement of the FreeBSD Core Team, I took over the role.

Continuity, and

Glen did an excellent job as a release engineer, so when I took over my priority was to provide continuity: To do things, as much as possible, the same way as he had done previously. To this end I was aided by three years of experience as Release Engineering Deputy Lead: While I had only managed one release before (FreeBSD 13.2-RELEASE, after Glen was hospitalized with pneumonia), I had watched how Glen did things enough to have a general sense of how to continue.

But that is not to say that I made no changes—if that were the case, this would be a very short article. My first goal was to streamline the release process to avoid a repeat of FreeBSD 13.2, which arrived almost a month late after needing six release candidates before all the issues were ironed out enough to proceed to the release. To that end, I set out some ground rules for release cycles:

- A Release Candidate should be exactly that—a candidate for a release—and so by the time we get to RC1 there should be no further known release-blocking issues. The goal is to build RC1, have a week of testing (or a bit under a week given the time for builds to complete), and then build the final RELEASE images. If problems are found in RC1 we can do a second or third release candidate—but this should only happen with new problems; any previously-known problems should have already been ironed out before RC1.
- While developers often have code that they "need" to get into the next release, I'm not going to hold up the release process if they're late—it's simply not fair to all the other developers who got their code into the tree on time and want to get it into the hands of users. The tree is open for anyone to merge patches until -BETA1 (we used to "freeze" the stable branch, but that hasn't happened for many years now) and between BETA1 and BETA2 I'm generally willing to accept changes that aren't particularly large or dangerous, but between BETA2 and BETA3 it becomes a matter of "are we *sure* this patch won't cause any new problems" and between BETA3 and RC1 the bar for patches rises to "... and does it fix an actual problem which users are running into" since if a problem is purely theoretical it's not worth the risk that something might be wrong with the patch. After RC1, of course, only the most critical patches will be accepted—and ideally

no patches—since any changes beyond that point will require adding another Release Candidate and pushing back the release date.

- If new features arrive close to the release—say, after we enter "code slush", two weeks before BETA1—and problems are found that can't be immediately and trivially fixed, *I will remove those features from the release*. Part of the reason that past release schedules often dragged on is that features arrived late and then needed multiple rounds of bug fixes before the release could ship; as with late-arriving code, it's not fair to everyone else if one developer is delaying the release because they merged buggy code at the last minute.

I also asked FreeBSD developers—with inconsistent success, although it does seem to be gradually improving—to be far more proactive in alerting the release engineering team to anything they intended to get into the upcoming release; and once I knew about issues, I became more proactive in emailing developers to ask for status updates. More than once I had replies along the lines of "I didn't realize BETA2 was already out; I'll get that code merged ASAP"—FreeBSD is a volunteer project so it's entirely reasonable that other things in developers' lives distract them from working on FreeBSD, but the last thing we want to do is have a release schedule slip just because someone lost track of time.

Once I knew about issues, I became more proactive in emailing developers to ask for status updates.

Release Scheduling

Once it became clear that the FreeBSD project could do releases in a predictable amount of time, I turned my mind towards scheduling. Three BETA images and one RC collectively take a month, and we have a two-week "code slush" before BETA1 and a two-week warning (aka "hurry and get your code merged") before that. That adds up to 2 months if everything goes perfectly; but since the schedule will *sometimes* slip no matter how hard we try (if nothing else, because we'll always hold the release for last-minute security fixes) we need a third month to allow for slack in the schedule—and to allow the release engineering team a bit of time between releases.

Knowing that we can effectively do one release per 3 months makes a schedule obvious: Do one release per calendar quarter. If we start with the 2 week "warning" and 2-week "code slush" in the first month of the quarter and have BETAs and RC1 weekly through the second month of the quarter, we end up with the RELEASE announcement landing around the start of the third month. If the release slips by a few weeks, we still finish before the end of the third month. This gives us a target schedule for future releases—we're never going to ship a release that isn't ready, but knowing what we're aiming for is an essential starting point:

- BETA1 - 28 days: Warning to developers.
- BETA1 - 14 days: Code slush starts.
- First Friday in second month of the quarter: BETA1.
- BETA1 + 7 days: BETA2.
- BETA1 + 14 days: BETA3.
- BETA1 + 21 days: RC1.
- BETA1 + 28 days: RELEASE builds start.

- BETA1 + 32 days: RELEASE announcement goes out.
- BETA1 + 39 days: The release branch is handed over to the security team.

The one exception to this is .0 releases: These start with ALPHA builds before the stable branch is created, and there's simply no way to fit the entire process for a .0 release into three months; so, in the interest of keeping the schedule otherwise aligned with calendar quarters, I decided we should set aside six months—two calendar quarters—for those releases. The exact schedule of when things will happen within those six months, I haven't decided yet—and since I've never done a .0 release before, will probably get wrong for 15.0—but I do hope to eventually establish a repeatable schedule for these as well.

Once the amount of time for each release was determined, all that remained was to set the order in which releases would occur and decide how often to do .0 releases. The first question was easily answered: There's no point doing a release 3 months after a release from the same stable branch, so we alternate between stable branches, i.e. 14.0, 13.3, 14.1, 13.4, 14.2, et cetera. The answer to the second question came from many discussions amongst FreeBSD developers over the years: We'd like to have a new major version every 2 years. In the past, it has worked out very well to have the .0 release cycle start soon after a FreeBSD developer summit—that ensures a large number of developers are on hand to discuss any features that we want to have completed in time for the release—and since the largest developer summit has typically been at BSDCan in May or June, it makes sense to schedule the .0 releases for the second half of the year, with the release landing around November or December of every odd year.

We'd like to have a new major version every 2 years.

This gives us a schedule for when each FreeBSD release will take place:

- June 2025: FreeBSD 14.3
- (September 2025: No release this quarter; working on 15.0)
- December 2025: FreeBSD 15.0
- March 2026: FreeBSD 14.4
- June 2026: FreeBSD 15.1
- September 2026: FreeBSD 14.5
- December 2026: FreeBSD 15.2
- March 2027: FreeBSD 14.6
- June 2027: FreeBSD 15.3
- (September 2027: No release; working on 16.0)
- December 2027: FreeBSD 16.0

Support Periods

Starting with FreeBSD 11.0-RELEASE, FreeBSD's policy for supporting releases has been that minor releases are supported for three months past the date of the next release from the same stable branch, while stable branches are supported for five years from the date of the .0 release.

The first half of that—support durations for minor releases—fits neatly with the new quarterly release schedule: Just as we have a new release at the end of nearly every quarter, a release reaches its end-of-life at the end of nearly every quarter. Moreover, since releases now always occur in the third month of each quarter, the policy of "three months of overlap between minor releases" simply translates to "one quarter of overlap."

Where the new release schedule doesn't fit with the established support timelines so well is the support period for stable versions: Doing a new .0 release (and thus a new stable branch) every 2 years and supporting each stable branch for 5 years would result in 3 stable branches being supported at the same time—which experience has taught us isn't something we can do very well as a project.

Consequentially, to better align support timelines with the release schedule the Core team, security team, and ports management team have approved adjusting the support timelines for stable branches: Starting from 15.x, stable branches will be supported for 4 years instead of 5. This may be less of a change than it sounds: In practice stable branches were never supported very well in their fifth year.

With this change, there are always two supported stable branches, aside from a window of a few weeks every second year where there are three supported branches when (N+2).0 is released shortly before the N.x branch reaches its end-of-life.

Starting from 15.x, stable branches will be supported for 4 years instead of 5.

Legacy Releases

For many years parts of the FreeBSD website—sometimes commented out—have referred to "legacy" releases, but as far as I could find there was never any clear definition of what this meant. During Glen's time as a release engineer, the concept mostly fell into disuse, as he felt that calling a release "legacy" gave users a misleadingly poor sense of its quality, when in fact we aim for the same quality from all FreeBSD releases.

I decided to bring this concept back to help users (especially new users) decide which release they should be installing. To this end, I have a definition:

> ***A "legacy" release is a release that has not yet reached its End of Life, but which the Release Engineering team recommends against deploying for new systems.***

In other words, it exists for the benefit of users who already have FreeBSD installed and don't want to upgrade to the latest release; but if you're installing a new system, you should probably be reaching for a newer release.

In general, once FreeBSD N.1 is released, the (N-1).x stable branch will be considered "legacy", and any time a new minor version is released from a stable branch, the previous version from that branch is immediately "legacy". Or put another way: Normally everything except the most recent release from the latest stable branch is "legacy", but immediately after a new stable branch launches with a .0 release the most recent release from the previous stable branch will remain "non-legacy".

Ultimately FreeBSD users can use whatever versions they like, and this is in no way intended to limit FreeBSD developers' decisions about merging features and bug fixes; it's purely a matter of helping new and inexperienced users pick the right version to download.

What's Next?

During 2024 I've made the changes mentioned above and managed the 13.3, 14.1, 13.4, and 14.2 releases. What comes next?

Well, for a start we have three releases scheduled for 2025: 13.5, 14.3, and 15.0. The minor releases I've done so far have each taken 50-100 hours of my time (13.x at the low end of the range and 14.x at the high end, since more new code lands in 14.x) and I've been very lucky to have sponsorship from Amazon for my FreeBSD work (both for maintaining FreeBSD specifically on the EC2 platform and for general release engineering work)—without that, FreeBSD 14.2 would have shipped without some late-landing features simply because it wouldn't have been possible for issues to get fixed in time. FreeBSD 15.0, of course, will be a new major version—something I've never done before—and I'm sure it will take considerably more release engineering time.

But beyond the "routine" process of pushing out weekly snapshots and (mostly) quarterly releases, there are two big items on the horizon: First, FreeBSD 15.0 should ship with a packaged-based system—pkgbase has been "coming soon" for far too long already—and second, the FreeBSD Foundation, with funding from the Sovereign Tech Agency, is funding a project to modernize the FreeBSD build process (in particular, to make it possible to build as much as possible without root privileges). Both items will involve significant changes in the release engineering process, but neither of them should affect our goal of producing stable and well-tested releases on a predictable schedule.

COLIN PERCIVAL is the FreeBSD Release Engineering Lead and maintainer of the FreeBSD/EC2 platform, for which he was recognized as an "AWS Hero" in 2019. His day job is as the founder and primary developer of Tarsnap, an online backup service.

Contents

GhostBSD: From Usability to Struggle and Renewal

BY ERIC TURGEON

This article isn't meant to be technical. Instead, it offers a high-level view of what happened through the years with GhostBSD, where the project stands today, and where we want to take it next. As you may know, GhostBSD is a user-friendly desktop BSD operating system built with FreeBSD. Its mission is to deliver a simple, stable, and accessible desktop experience for users who want FreeBSD's power without the complexity of manual setup. I started this journey as a non-technical user. I dreamed of a BSD that anyone could use.

The Beginning of GhostBSD

In 2007, I read Eric S. Raymond's *How To Become A Hacker*. It pointed to BSD Unix as a place for aspiring contributors to learn and grow. It sparked my curiosity about BSD, which led me to explore FreeBSD. At that time, I was using Ubuntu, and the question emerged: could FreeBSD become a desktop OS for non-technical users like Ubuntu? At that time, I was a non-technical user myself. I just liked Ubuntu's ease and wondered why FreeBSD couldn't be the same. In 2008, that question ignited my quest to create GhostBSD, and I started to learn everything I needed to create the project to make my vision of FreeBSD as approachable as Ubuntu.

I started this journey as a non-technical user. I dreamed of a BSD that anyone could use.

It took me almost two years to figure everything out, experimenting with tools like FreeSBIE to craft a live CD and finding some code to build with GNOME. FreeSBIE was tricky for a beginner to wrangle. As a Canadian French speaker, the *FreeBSD Handbook* was helpful, but my English was limited, and I forced myself to learn it. I dug through forums and docs, piecing together GNOME builds, often breaking thighs more than making things work. Fun fact: the name "GhostBSD" comes from my wife. Back then, she was my girlfriend. She suggested it, and I went with it. We got married in September 2009, just as the project took shape. In November 2009, GhostBSD 1.0 Beta launched as a live CD running GNOME on FreeBSD 8.0. It was rough and full of issues, but it laid the foundation for what GhostBSD has become. Feedback from the FreeBSD community fueled the early progress. Along the way, some people joined to help and teach me about source control versioning and other things. Folks like Ovidiu Angelescu nudged me toward SVN with his shell scripting know-how. I did learn a lot. We use Git now, but back then, it was all SVN.

The Early Years

The first release was GhostBSD 1.0 Beta, which gave users a taste of FreeBSD with GNOME. It was a shaky start but a proof of concept. By 2010, version 1.5 added a text-based installer using pc-sysinstall from PC-BSD, making setup easier. Those early years involved learning FreeBSD in and out, shell scripting, programming, and from people like Ovidiu Angelescu. I leaned on Ovidiu for shell scripting tips. In 2011, version 2.5 introduced the graphical GBI installer, built on the same pc-sysinstall backend. That backend remains a core component today. Its reliability meant I didn't have to reinvent the wheel. It was a lesson in sticking with what works.

The groundwork for NetworkMgr started around 2012 as a GUI tool to manage Ethernet and Wi-Fi connections. It was a step toward usability, inspired by Linux's NetworkManager. By 2015, I stripped it from ghostbsd-build to refine it separately as its tool. A significant shift hit in 2013 with version 3.5. GNOME 3's release was clunky and unstable on FreeBSD. It was laggy and resource-hogging. This situation clashed with GhostBSD's goals. We tried other desktop environments. We created multiple ISOs with different DEs like LXDE, XFCE, and Openbox, and the meaning of GhostBSD, "GNOME hosted on BSD" faced an identity crisis. When MATE, a fork of GNOME 2, emerged, it saved the day. We did switch to MATE for its simplicity and familiarity. It felt like GNOME 2's cozy and at home, running lighter on FreeBSD. At that time, I was unsure what to do with the other desktops when some people just disappeared from the project. I started dropping all the other desktops, but XFCE remained an option. MATE became the flagship, reshaping GhostBSD's focus on usability over Flash.

MATE became the flagship, reshaping GhostBSD's focus on usability over Flash.

System Base Shifts

I started working on Update Station in 2014 to bring GUI updates to GhostBSD users. I thought it was later, but after digging through GitHub, I found that it began taking shape around that time. At first, GhostBSD relied on FreeBSD's release source, tarballs, and official packages. At one point, we needed to start delivering updates for tools like NetworkMgr, which pushed us to build our package repositories. Our custom packages often clashed with FreeBSD version upgrades, creating friction and demanding a better solution. Also, freebsd-update was hard to automate for Update Station. Freebsd-update didn't fit our GUI-first goal. We started to look at what TrueOS was doing. I noticed the use of PkgBase, and it was intriguing. Their pkg-driven OS updates promised GUI freedom.

In 2018, GhostBSD 18.10 switched to TrueOS as its base. TrueOS offered PkgBase, letting us update the OS with the pkg tool, and we ditched freebsd-update. It also brought OpenRC, a modern service manager perfect for building a GUI to manage services, but that never happened. The shift to TrueOS allowed Update Station to upgrade software and OS from a graphical interface. It was a game-changer for us and allowed our users to upgrade the OS with Update Station. Later, TrueOS introduced OS ports, allowing us to build OS packages from the ports tree with poudriere and giving us finer control over updates.

Bumps in the Road

TrueOS shut down in 2020, putting pressure on us to maintain everything we'd gained. I initially wanted to keep OpenRC, but maintaining all services across ghostbsd-ports and ghostbsd-src became a solo struggle. Around that time, I was alone, primarily maintaining it, which was draining my time to improve and manage GhostBSD. In 2022, I dropped OpenRC for FreeBSD's simpler but reliable RC system. It meant less to juggle and more focus forward. By 2023, maintaining OS ports grew too taxing. In 2024, I decided to shift building our OS packages from FreeBSD-maintained PkgBase within SRC, streamlining the maintenance load and refocusing on user experience.

In January this year, I realized that building GhostBSD from STABLE was too much for our small team. After discussing it with the other contributors, I decided to switch back to FreeBSD RELEASE. Yes, we are losing early driver access, but we save time debugging STABLE changes, giving us the stability to build on. I am not saying that STABLE is always broken, but sometimes, some changes create problems.

I feel we're in a good place to focus on improving our tools.

Over the years, I've tried to manage GhostBSD to the best of my knowledge through critical choices that added difficulties but also brought gains for what was missing in GhostBSD. PkgBase and OS ports gave OS updates from a graphical user interface, but OpenRC piled on work. However, it also meant adding more to maintain than the project could handle. All the latest changes mark the return to GhostBSD's roots and a renewal of focusing on usability instead of over-complicating the maintenance of GhostBSD. It was a hard lesson to keep it simple.

Where GhostBSD Stands Presently

As I write this, we've just released GhostBSD 25.01-R14.2p1. It marks a shift from FreeBSD STABLE to FreeBSD RELEASE, using 14.2-RELEASE-p1 for more excellent stability. The new versioning of GhostBSD breaks down as follows: 25 for 2025, 01 for the GhostBSD patch, R for RELEASE, 14.2 for the FreeBSD version, and p1 for the FreeBSD patch. It aims to clarify releases for users. No more guessing what's what. With all the recent changes, I feel we're in a good place to focus on improving our tools.

Some of those tools are:

- **NetworkMgr:** A GUI software for Ethernet and Wi-Fi, modeled after Linux's Network-Manager. Simple clicks over CLI chaos.
- **Update Station:** A GUI software for software and OS upgrades that creates a Boot Environment backup before upgrading. Safety first!
- **Software Station:** A GUI software to install software that leverages pkg to install software. The point, click, done.
- **ghostbsd-build:** Used to build GhostBSD, including Joe Maloney's ZFS reroot hack for a read-write ZFS pool live session in RAM. Blazing fast for demos and installations.
- **Backup Station:** Added in September 2022 by Mike Jurbala, a GUI software that uses pybectl, an in-house Python module for interfacing with bectl, to manage Boot Environments. System snapshots made easy.

- **GBI and pc-sysinstall:** GhostBSD's GUI installer recently dropped UFS in the UI to leverage ZFS's strengths. ZFS's power shines over older ways.

What's Next for GhostBSD Future

For 2025, I plan to document some SOPs to make contributing to GhostBSD easier for everyone, hoping that new contributors won't need too much mentorship. I'll deploy a faster build server at my house for quicker package builds and as I write this, I am waiting for the PDU to arrive. I also want to finish an OEM-friendly installer to widen our reach, rework Update Station to install updates at boot, and improve NetworkMgr's integration with devd for stability. If possible, add support to create a home directory dataset with the option to encrypt with GBI. FreeBSD's upcoming AC and AX Wi-Fi support in 2025 or 2026 promises better connectivity speed, and I'm excited about that. Our laptops will love it.

I can't speak for other contributors, but we have a long list of tasks on GitHub. We do have a roadmap that can be found under the Development tab on GhostBSD.org. Check it out if you're curious.

In the long term, we are waiting for more donations to come in so we can buy an ARM(Ampere) server to start building GhostBSD arm64. In the meantime, GhostBSD still aims to be a fully GUI-driven OS that leverages ZFS, which is perfect for non-technical users who can benefit from what FreeBSD offers. We have conversations on creating components for a desktop to slowly replace MATE that align better with FreeBSD/GhostBSD, like a file manager that leverages ZFS. However, it's still just discussions. Nothing solid for the moment.

GhostBSD has been a journey with a chain of choices, from a 2009 live CD to what it has become today.

Conclusion

GhostBSD has been a journey with a chain of choices, from a 2009 live CD to what it has become today. It was started by a non-technical user dream and evolved into a community project. Each step, like custom packages, TrueOS, ZFS reroot live session, and PkgBase tackled a challenge. The past taught me resilience, the present offers stability, and the future invites you to help shape it. If you are interested in getting involved, please visit GhostBSD.org. You'll find a spot.

ERIC TURGEON is the founder and leader of GhostBSD. He's also a FreeBSD ports committer, focusing on maintaining MATE ports and the NetworkMgr port. Based in Canada, Eric has been passionate about BSD since the late 2000s. He balances his time between GhostBSD, FreeBSD contributions, work, and personal life. His drive comes from a desire to make BSD accessible, and he welcomes anyone to join the GhostBSD community.

Contents

BSD Now and Then

BY BENEDICT REUSCHLING

The BSD Now podcast recently celebrated its 600th episode, which seems like a perfect opportunity to give FreeBSD Journal readers a behind-the-scenes look at this long-running BSD show.

Humble Beginnings

BSD Now started in 2013 as a podcast hosted by Allan Jude and Kris Moore. I was a regular listener back then, excited to hear the news in the BSD space. That space quickly became "the place to B..SD" as Kris so famously quipped. The show offered both news and tutorials, the latter of which Allan recorded separately. He once told me that this was difficult because you had to both type and explain what you were doing. Typos or other unexpected computer glitches meant either undoing the changes and cutting the recording or starting over altogether. From the beginning, the show has offered a feedback channel via email for the submission of ideas, show content, or discussions about anything in the BSD space, and the feedback is read at the end of the episodes. Often, people use it to ask questions about installing or using BSD. Sometimes, Allan has offered his vast knowledge of ZFS to help users build their NAS at home or understand difficult-to-understand concepts.

From the beginning, the show has offered a feedback channel via email.

Kris also offered his perspective from the PC-BSD side and everything about modern Unix desktops. This combination contained all the good bits I was looking for as a BSD user. Since you only had to wait a week for the next dose of BSD Now, it meant that my BSD batteries, which always became supercharged after a conference, would not deplete so fast. This went on for several years and this brief description of the early days of the show does not do it justice—and Allan and Kris could tell many more stories from that time!

I vividly remember when BSD Now became something more for me than just a podcast I listened to. We were at FOSDEM and had gone as a group to one of the many good restaurants in Brussels. As we sat at a long table waiting for dinner to arrive, Allan mentioned that Kris Moore intended to step down from the podcast as his life was getting busier. They were looking for a replacement moderator. Hearing this, I innocently asked what would be involved in that. I sometimes cannot keep my mouth shut and at that point had not even considered stepping in. Allan explained the process briefly, how they would meet for an hour each week for recordings, and some of the technical logistics behind the show. This all sounded exciting, and as I was always willing to help when I could, I said that I'd be willing to join. Allan liked that and we continued our discussion on other channels (also: dinner had arrived).

The change was not announced until we had done a recording in a separate room at the AsiaBSDCon venue on March 16, 2017. This handover episode 185 gave Kris a chance to say goodbye and me to introduce myself. These early episodes proved to be a bit bumpy, but eventually, I got the hang of it. It was good to practice my English every week and keep up to date on the BSD space at the same time.

Becoming Independent

BSD Now was running as part of the Jupiter Broadcasting podcast network. It meant some cross-pollination between the shows, as listeners would likely also taste the other shows available under the same umbrella. Also, Allan has been a popular figure since his TechSNAP days, which probably boosted the listenership. The network also provided a lot of logistics like post-production. For them, it was just another episode to cut into shape and publish in time, which allowed us to focus entirely on the recording and content.

BSD Now was running as part of the Jupiter Broadcasting podcast network.

The podcasting world has seen some mergers and takeovers. In the fall of 2018, it was announced that Linux Academy would merge with Jupiter Broadcasting. In turn, a company named A Cloud Guru bought Linux Academy. With the typical restructuring of operations these mergers often encompass, we would still run the show under the new company name. Allan was well connected to the original Jupiter Broadcasting people like Chris Fisher and Bryan Lunduke who launched the network in 2008. In episode 347 (April 23, 2020), we announced that we would become independent of Jupiter Broadcasting. (*Chris Fisher would make Jupiter Broadcasting an independent operation again in August 2020*). This involved changes, and much like becoming a self-publishing author, it meant taking care of a lot of managerial items around the show. Lucky for us, we retained the talent of Angela Fisher for behind-the-scenes work like posting the finished episodes to the many podcast channels. And we still managed to put out weekly episodes like before. JT, our producer started doing production at about episode 100. He had been the producer and helped with production work of the Linux action show (LAS) since 2013. Chris Fisher decided to take the show in a different direction and didn't need his assistance in that role anymore. At about the same time Allan was looking for a replacement for the original producer TJ. Due to JT's involvement in development of the lumina desktop with Ken Moore and working on PC-BSD, Allan had asked him if he was interested in replacing TJ. That's how he became our new producer and has been ever since. He would prove to be an invaluable help in doing the hour-long post-production editing, and that allowed Allan and me to spend our regular hour per episode without extra overhead. While we never relied on click-through rates or other viewer-dependent marketing to finance the show, it was still necessary to compensate people for their efforts. For the longest time, Colin Percival's Tarsnap has been sponsoring the show, and we never cease to praise the backup service he runs. This regular sponsorship allows us to continue our little podcast as an independent operation. We have always been thankful for his continued and generous sponsorship!

How the Sausage Is Made

An episode starts with the two moderators agreeing on a recording date. Once we've found a convenient time, we let our producer, JT, know about it. He then prepares the show notes scaffolding in a shared document. We all contribute stories to BSD Now and send them to the producer. Sources vary from stories we find on tech news sites, forums, individual blog posts, submissions by listeners via email, and the usual suspects. The latter are sites that regularly put out content we cover (in the length that we prefer) and that are either BSD-focused or have a history of independently covering the BSDs well.

JT then tries to fit it into a show episode based on some criteria: newsworthiness, actuality (breaking news), long and shorter items, and who is recording next. This last bit is important because some of us moderators are more versed in certain topics (*except for me…maybe I'm there to keep things balanced!*) and can provide extra detail during that piece in the show. Longer news items with current events tend to become headlines, while shorter ones either get a spot in the news roundup or even the beastie bits. It all needs careful curation to fit into the 45-minute time slot. Longer recordings need cutting when a story tends to overgrow the episode. To prevent that, we aim for the 45-minute mark. Interviews and special episodes go somewhat longer.

Before the recording starts, we moderators agree on who will cover which items in the show

When an episode stub is filled with items and posted to our ever-growing show doc dumping ground, I fill out some metadata. This includes the recording date, tags based on the news items in the show, and the title of course. Yes, you can blame me for all the nonsensical episode titles that make you roll your eyes when an episode pops up in your podcatcher. Back in the early days of the show, there used to be a rule that the episode title had to be made up of three words. Not sure where this limitation came from, but maybe marketing had a say in it (catchy title and all). This is both a blessing and a curse: keeping the title short ensures we don't start with a whole sentence and short is easy to remember. On the flip side, it's not always easy to come up with a cool title based on the content of the show. For example, if there were a story about NetBSD running a big ferris wheel, what would you call such an episode? "Wheel of Fortune?", "Big wheel NetBSD," or perhaps "Wheeling NetBSD"? I usually spend a bit of time inventing something. (None of my co-moderators have ever protested or even demanded I change the title before recording.)

Once all that is in place, we have our show notes ready to record an episode. Ideally, the notes are ready in time to let everyone look at the stories and prepare. But life and circumstances sometimes don't allow that, and we read a story on air for the first time. That's not ideal but does allow for a fresh reaction when reading the content.

On Air

Before the recording starts, we moderators agree on who will cover which items in the show, and that lets each of us cover our favorite part. We typically take turns going through the material. This has worked well as it gives the other person time to recover from a long read and even prepare for the next article.

We moderators can see each other during the recording session and that has proven to be invaluable. We can see if someone is finishing up their piece soon. Giving nods, hand signals or even holding up items we're currently discussing are all informative for the other person. Remember, we are not sitting in the same room during recording. Also, we are not in a professional recording studio. Even though we use the same audio equipment for recording, a smooth-running show can still turn bad during a recording. Loss of Internet connection, construction work outside the window, airplanes, or delivery men ringing our doorbells during recording have all caused interruptions in the past. Luckily, these events are not common. Our producer can fix some audio glitches during post-production, but he can't do magic. If things go bad, we must re-record a bit or (worst case) a whole episode! This is tough. Since we do all the episodes from top to bottom without stopping, we can not easily insert missing pieces or recreate the exact words we spoke. The audience would notice those things as they would not sound natural.

Special episodes are the ones we do as interviews with people in the larger BSD space.

Audio troubles have been a common occurrence and sometimes we don't realize it until listeners tell us about it. It could have been that one of us was too silent, or that the audio tracks shifted together, and we talked over one other. Microphone settings may change unexpectedly. I recently realized that the headphones I had used since my first episode had died. At first, I thought the cause was a bad volume setting on the other end. After testing, I found that one side went dead. I have since switched headphones and can hear my co-moderators as clearly as if they were standing next to me. Audio can be a tricky business!

Specials

Allan did a lot of podcasting, even before BSD Now we do as interviews with people in the larger BSD space. We also did some recordings during conferences where we could conveniently drag someone into a quiet corner to ask them some questions. This is both exciting and unpredictable as we can't create a show notes document up-front. We do write up a couple of questions when we are interviewing our usual recording spots, but during a conference, we need to come up with them as we go along depending on the interviewee. I am always glad that people are willing to do this. They reach a wider audience through us and can talk about what they are working on, call for testers, and provide some background info on how they got started with the BSDs. That is the question Allan and Kris always asked as the first question and we keep that tradition alive.

I was intrigued by the interview Kris and Allan did with Bryan Cantrill in episode 103. I had never heard about him, but I replayed that interview multiple times as an evergreen because of the Unix backstories and rants. A lot of listeners shared this excitement, so it was prudent for them to do a follow-up interview in episode 117. We had hit a nerve and looking back from where we are now in the IT sector, it is amazing to see some parallels and histories repeating.

Other people were kind enough to be interviewed by us, offering their unique experiences and expertise. Michael W. Lucas became a regular and must have been the most interviewed person over the years. Not only do we cover his blog posts about his progress in book writing, but we also love interviewing him when a new book becomes available. Michael's interviews offer a fascinating insight into the self-publishing business. The written and spoken word combine to each other's benefit!

We also had guest moderators from time to time. This happened when we could foresee that one of us needed a longer break (because life happens to everyone), so someone else had to take over. I did at least two episodes with Dan Langille of BSDCan-fame—just one of his many achievements. Episode 404 (BSD Now moderators not found) was super special in that all moderators were replaced. This had to be planned in advance because we needed to find and convince replacement moderators. Our producer, JT, managed that and seemed to have a lot of fun producing the special.

Other special episodes are those where the counter hits a certain number. Geeky numbers like 386, 486, and others deserve something special. I'm a bit worried about episode number 666 but look forward to 686. We recently talked about what we should do once we hit 999. Maybe call it a day and announce the end of the podcast after such a long time. We might as well keep going past the big 1k episode—the sky is the limit and it's too early to tell.

Scheduling episodes is not always easy. Holidays, conference visits, unexpected sicknesses, or other things happen. These days, we keep a buffer of one episode to cover for a week of absences. For holidays like Christmas or New Year, we usually plan something special and record that up-front. This extra work during a busy holiday season is offset by the time it buys to spend our usual recording day with friends and family.

Comings and Goings

Our lives became busy, and Allan needed a break from weekly recording. First, we switched to recording two episodes per week, giving us free time the other week. As time went on and the busyness increased even more, we brought in Tom Jones at episode 400 (see the pattern?) as a third moderator. This meant that both Allan and Tom had to record only once per month, while I remained the constant element every two weeks. Tom provided his network expertise to the show and was able to comment on these topics more than I did.

Allan did a lot of podcasting, even before BSD Now.

Allan did a lot of podcasting, even before BSD Now. In an interesting twist, he launched the 2 1/2 admins podcast in his off week. All of us would not enjoy the weekly BSD news if it had not been for Allan and his courage in trying the experiment of a BSD podcast. As the last member of the original BSD Now moderators, he told us in 2023 that he would leave the podcast to make time for his company Klara Systems. In episode 512 (patterns, patterns), recorded live in front of an audience during BSDCan in June 2023, he announced this decision. To keep the show going, he brought in Jason Tubnor, who had done podcasting before. Allan is still sticking around our private production channel and still offers his help and perspective if needed. Our producer, JT, became the new mastermind behind the scenes, both in the pre-and post-production, as well as the business side of the show. He launched our new Patreon donation levels, put up all the past episodes, managed the website and so much more.

Jason brings a lot of experience in both OpenBSD and FreeBSD to the show. Often, he provides extra info on certain topics, be it security, running BSD in production, or virtualization with bhyve. I look forward to chatting with either Tom or Jason before the show starts. These catch-ups were often so informative, that we decided to record them as well for our higher-level Patreon sponsors. This includes our current projects, insights, mini-rants, upcoming events, and other dealings. After that, we start the official show recording as we have done so many times before.

Audience Matters

Producing an episode every week is still an amazing achievement, and the audience expects a new episode every week. People have told us they are hopelessly behind in listening to episodes or just jump ahead a few weeks. Others send in their thoughts about pieces we covered or follow up on a question someone posed that we could not answer. We have had surprised blog authors thanking us for covering their online writing. For me, covering the blogs is rewarding in that it gives us content and the blog some extra traffic it would not otherwise get. It may also encourage them to continue blogging about BSD or Unix in general, which will then re-appear in a future episode. Who knows, some good collaboration may start when a listener gets in touch with the blog author about something they wrote about.

Producing an episode every week is still an amazing achievement.

On more than one occasion, people have recognized me from hearing my voice. Be it at BSDCan where people I've never met before approached me to say "hi" or the occasional student in my University mentioning the podcast. It's great to start conversations like that and I usually want to know what they like about the podcast. We're far from the celebrity status and I don't know if I would want that life, but there are people out there who look forward to our show every week and appreciate the efforts we put into it. Feedback emails keep us going. It is a rewarding experience whose value is not about how many listeners we have, but the individual benefits everyone draws from it. We have answered a whole lot of technical questions over the years and I'm sure it helped spread the use of BSD.

Future Dealings

People often ask us why we do not bring the video back they did at the beginning of the show. For one, it is difficult to lip-sync a video when doing editing and doubles the effort when not just showing our faces, but also websites or even videos we cover. JT would have to bear the brunt of the editing work. For us moderators, it would be less of a problem and our software does record the video during the recording sessions. Some listeners have also commented that the video does not give them any added value. Be it the shape of my face or the fact that they listen to the audio while commuting to work (and 45 minutes seems to be a sweet spot), video is not seen as a benefit to some. As a compromise to those who like to see us, we have recently added the raw video as an option for our Patreon sponsors. This is as raw as it can get without any editing, but if you want talking heads, they are there (for a price).

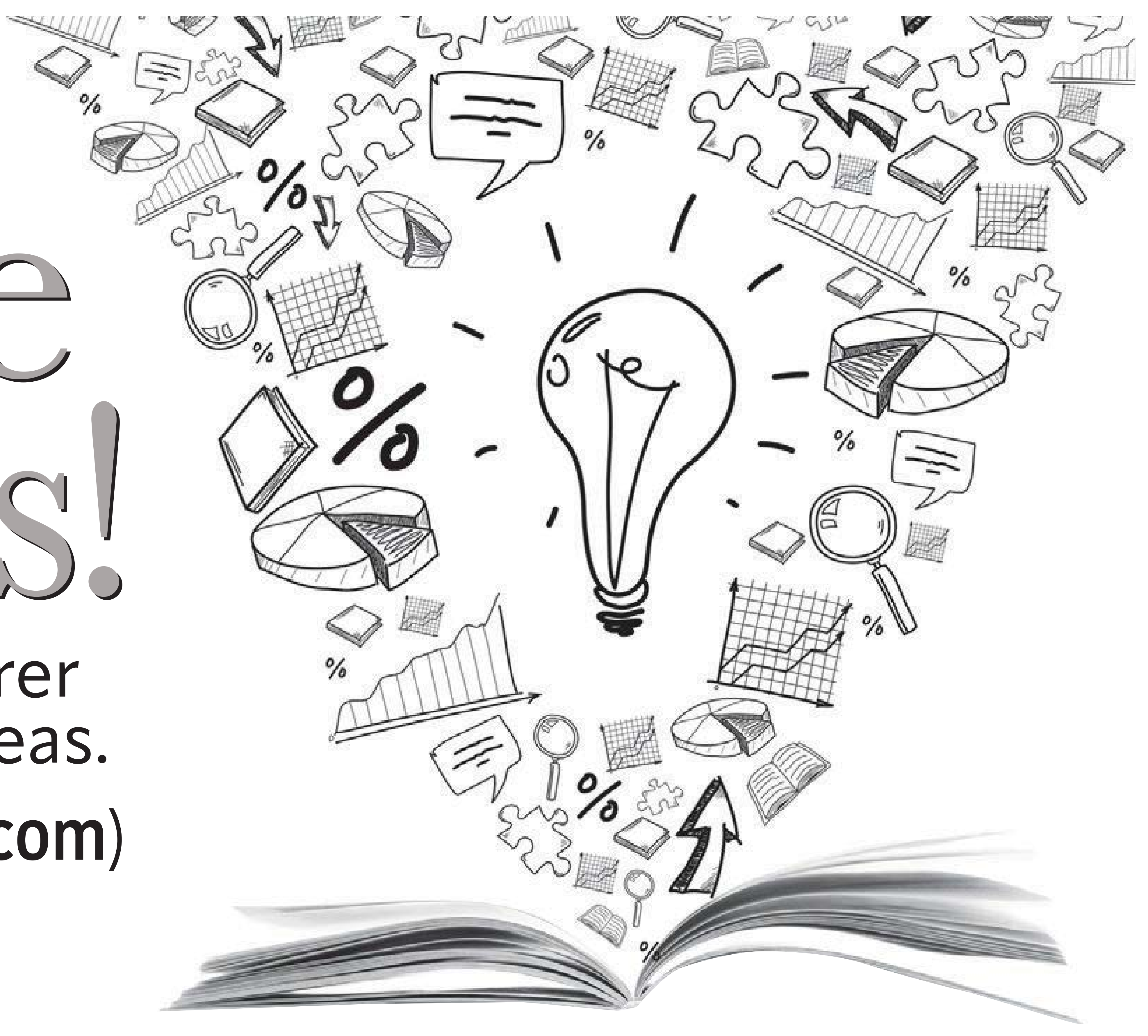
Will there be other changes, either in the way the show runs or who is speaking into the microphone? We do not have any specific plans, but as the history of the show has shown, we sometimes must adapt to changing times and circumstances. The show is unique, just as the BSDs are. Podcasts remain as popular as ever and we seem to have carved out our little space. All of this is thanks to our sponsors and the listeners, who have been with us for over 600 episodes. Without people blogging, writing code, doing events, and writing articles about the BSDs, we would have nothing to report. I am excited about what we will be covering each week and how it will be received by the audience. If that does not change, then this remains each and everyone's place to be...SD.

BENEDICT REUSCHLING is a documentation committer in the FreeBSD project. In the past, he served on the FreeBSD core team for two terms and was vice president of the FreeBSD Foundation. For more than 8 years, he administered a big data cluster at the University of Applied Sciences, Darmstadt, Germany. He's also teaching a course "Unix for Developers" for undergraduates. Benedict is one of the hosts of the weekly bsdnow.tv podcast.

Write For Us!

Contact Jim Maurer with your article ideas.

(**maurer.jim@gmail.com**)



Contents

Support FreeBSD®

Donate to the Foundation!

You already know that FreeBSD is an internationally recognized leader in providing a high-performance, secure, and stable operating system. It's because of you. Your donations have a direct impact on the Project.

Please consider making a gift to support FreeBSD for the coming year. It's only with your help that we can continue and increase our support to make FreeBSD the high-performance, secure, and reliable OS you know and love!

Your investment will help:

- Funding Projects to Advance FreeBSD
- Increasing Our FreeBSD Advocacy and Marketing Efforts
- Providing Additional Conference Resources and Travel Grants
- Continued Development of the FreeBSD Journal
- Protecting FreeBSD IP and Providing Legal Support to the Project
- Purchasing Hardware to Build and Improve FreeBSD Project Infrastructure

Making a donation is quick and easy.

freebsdfoundation.org/donate

FreeBSD™ FOUNDATION

Character Device Driver Tutorial (Part 3)

BY JOHN BALDWIN

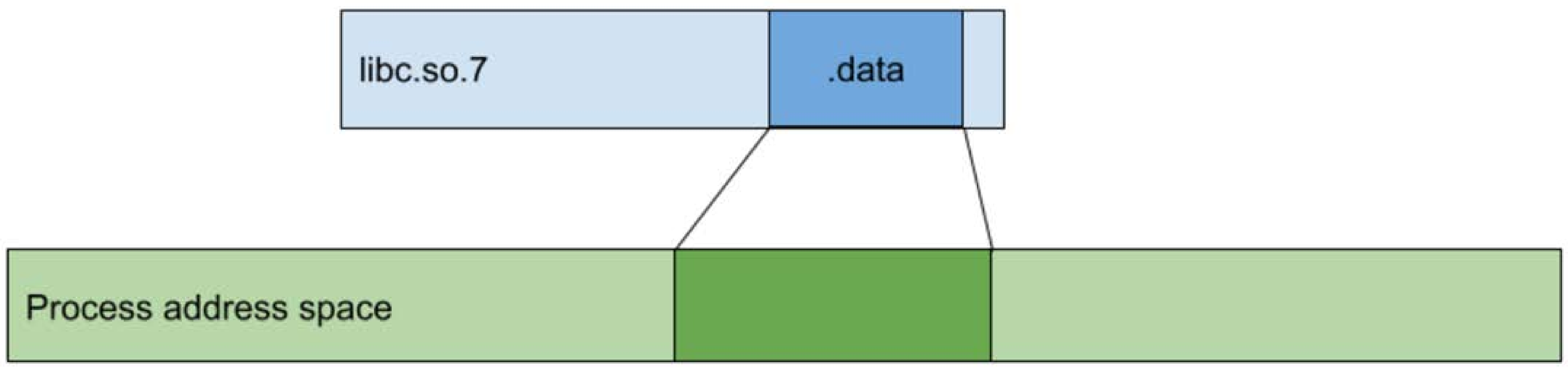
In Part 1 and Part 2, we implemented a simple character device driver that implemented support for basic I/O operations. In this final article in this series, we will explore how character devices can provide backing store for memory mappings in user processes. Unlike the previous article, we will not be extending the echo device driver but will instead implement new drivers to demonstrate memory mapping. These drivers can be found in the same repository as the echo driver at https://github.com/bsdjhb/cdev_tutorial.

Memory Mappings in FreeBSD

To understand how memory mapping works in character devices, one must first understand how the FreeBSD kernel manages memory mappings in general. FreeBSD's virtual memory subsystem is derived from the Mach virtual memory subsystem inherited from 4.4BSD. While FreeBSD's VM has seen substantial changes over the past thirty years, the core abstractions remain the same.

In FreeBSD, a virtual memory address space is represented by a virtual memory map (`struct vm_map`). A VM map contains a list of entries (`struct vm_map_entry`). Each entry defines the properties for a contiguous range of address space including the permissions and the backing store. Virtual memory objects (`struct vm_object`) are used to describe the backing store for mappings. A VM object has its own logical address space of pages. For example, each regular file on disk is associated with a VM object where the logical address of a page in the VM object corresponds to offsets within the file, and the contents of a logical page are the file contents at the given file offset. Each VM map entry identifies its backing store as a range of logically contiguous pages starting at a specific offset from a single VM object. Figure 1 shows how a single VM map entry can be used to map the .data section from the C runtime library into a process' address space.

Figure 1: Mapping of C Runtime Library .data Section



Each VM object is associated with a pager which provides a set of functions used to determine the contents of the pages associated with a VM object. The vnode pager is used for VM objects associated with regular files from both block-storage filesystems and network filesystems. Its functions read data from the associated file to initialize pages and write modified pages back to the associated file. The swap pager is used for anonymous VM objects that are not associated with a regular file. Zero-filled pages are allocated on first use. If the system runs low on memory, the swap pager writes less frequently used dirty pages to swap until they are needed again.

Logical pages in VM objects are represented by a VM page (`struct vm_page`). During boot, the kernel allocates an array of VM pages such that each physical page of RAM is associated with a VM page object. VM pages are mapped into address spaces using architecture-specific page table entries (PTEs). Managed VM pages maintain a linked list of mappings using architecture-specific structures called PV entries. This list can be used to remove all the mappings for a VM page by invalidating the associated PTEs so that a VM page can be reused to represent a different logical page, either for a different VM object or a different logical page address within the same VM object.

4.4BSD included a device VM pager to support character device memory mappings.

Each invocation of the mmap(2) system call creates a new VM map entry in the calling process. The arguments to the system call provide various properties of the new entry including the permissions, length, and offset into the backing VM object. The file descriptor argument is used to identify the VM object to map into the calling process' address space. To map memory from a character device, a process passes an open file descriptor for the character device as the file descriptor argument to the `mmap()` system call. The role of the character device driver is to decide which VM object is used to satisfy a memory mapping request as well as the contents of the pages backing the VM object.

Default Character Device Pager

4.4BSD included a device VM pager to support character device memory mappings. This device pager is designed to map regions of physical memory that do not change while the OS is running. For example, it can expose MMIO regions like a frame buffer directly to userspace.

The device pager assumes that each page in a device VM object is mapped to a page of physical address space. This page can be a page of RAM or associated with an MMIO region. Importantly, once a logical address in a device VM object is associated with a physical page, that mapping cannot be changed. This assumption works both ways in that the device pager also assumes that once a page of physical address space is associated with a device VM object, that physical page can never be reused for any other purpose. As a result, the VM pages used by the device pager are unmanaged (no PV entries). However, this also means that the VM system is not easily able to find existing mappings of these VM pages to revoke existing mappings. In particular, destroying a character device via `destroy_dev(9)` does not revoke existing mappings.

The default character device pager uses the character device mmap method both to validate mapping requests and to determine the physical address associated with each logical page address. The mmap method should validate the offset and protection arguments. If the offset is not a valid logical page address or the requested protection is not supported, this method should fail by returning an error code. Otherwise, the method should store the physical address for the requested offset in the physical address argument and return zero. If the page should be mapped with a memory attribute other than `VM_MEMATTR_DEFAULT`, the memory attribute should be returned on success as well. When a mapping is created, the device pager invokes this method on each logical page address of the requested mapping to validate the request. For the first page fault of a logical page address, the device pager invokes the mmap method to obtain the physical address and memory attribute of the backing page.

Listing 1 shows the mmap method for a simple character device driver that uses the default device pager. This device allocates a single page of RAM when loaded and saves a pointer to this page in the `si_drv1` field. Due to the limitations of the character device pager, this driver cannot be unloaded. Example 1 demonstrates a few interactions with the device once it is loaded using a `maprw` test program to read and write from a mapping of the device.

Listing 1: Using the Default Device Pager

```
static int
mappage_mmap(struct cdev *dev, vm_ooffset_t offset, vm_paddr_t *paddr,
    int nprot, vm_memattr_t *memattr)
{
      if (offset != 0)
            return (EINVAL);

      *paddr = pmap_kextract((uintptr_t)dev->si_drv1);
      return (0);
}
```

Example 1: Using the /dev/mappage Device

```
# maprw read /dev/mappage 16 | hexdump
0000000 0000 0000 0000 0000 0000 0000 0000 0000
0000010
# jot -c -s "" 16 'A' | maprw write /dev/mappage 16
# maprw read /dev/mappage 16
ABCDEFGHIJKLMNOP
```

Mapping Arbitrary VM Objects

Due to the limitations of the default character device pager, FreeBSD has extended the support for character device memory mappings. FreeBSD 8.0 introduced a new mmap_single character device method. This method is called on every `mmap()` invocation that maps a character device. The mmap_single method must validate the entire `mmap()` request including the offset, size, and requested protection. If the request is valid, the method should return a reference to a VM object to use for the mapping. The method can either create a new VM object or return an additional reference to an existing VM object. If the mmap_single method returns the `ENODEV` error (the default behavior), `mmap()` will use the default character device pager.

The mmap_single method can also alter the offset (but not size) used for the mapping when returning a VM object. This permits a character device to use the initial offset of a mapping as a key to identify a specific VM object to map. For example, a driver might have two internal VM objects and use offset 0 to map the first VM object, and an offset of `PAGE_SIZE` to map the second VM object. For the second case, the mmap_single method would reset the effective offset to 0 so that the resulting mapping starts at the beginning of the second VM object.

However, a character device doesn't have to use multiple VM objects to benefit from the mmap_single method. The ability to use VM objects with other pagers can be useful. For example, the physical pager creates VM objects backed by wired pages of physical RAM. Unlike the default device pager, these pages are managed and can be safely freed when the VM object is destroyed. Listing 2 updates the mappage device driver from earlier to use a physical pager VM object instead of the default character device pager. This version of the device driver can be safely unloaded since the VM object will persist after the driver is unloaded until all mappings have been destroyed.

Listing 2: Using the Physical Pager

```
static int
mappage_mmap_single(struct cdev *cdev, vm_ooffset_t *offset, vm_size_t size,
    struct vm_object **object, int nprot)
{
      vm_object_t obj;

      obj = cdev->si_drv1;
      if (OFF_TO_IDX(round_page(*offset + size)) > obj->size)
            return (EINVAL);

      vm_object_reference(obj);
      *object = obj;
      return (0);
}

static int
mappage_create(struct cdev **cdevp)
{
      struct make_dev_args args;
      vm_object_t obj;
      int error;

      obj = vm_pager_allocate(OBJT_PHYS, NULL, PAGE_SIZE,
          VM_PROT_DEFAULT, 0, NULL);
      if (obj == NULL)
            return (ENOMEM);
      make_dev_args_init(&args);
      args.mda_flags = MAKEDEV_WAITOK | MAKEDEV_CHECKNAME;
      args.mda_devsw = &mappage_cdevsw;
```

```
	args.mda_uid = UID_ROOT;
	args.mda_gid = GID_WHEEL;
	args.mda_mode = 0600;
	args.mda_si_drv1 = obj;
	error = make_dev_s(&args, cdevp, "mappage");
	if (error != 0) {
		vm_object_deallocate(obj);
		return (error);
	}
	return (0);
}

static void
mappage_destroy(struct cdev *cdev)
{
	if (cdev == NULL)
		return;

	vm_object_deallocate(cdev->si_drv1);
	destroy_dev(cdev);
}
```

Per-Open State

In the first article in this series, we demonstrated support for per-instance data using the **`si_drv1`** field. Some character device drivers need to maintain a unique state for each open file descriptor. That is, if a character device is opened multiple times, the driver wishes to provide different behavior to each open reference.

FreeBSD provides this feature via a family of functions. Typically, a character device driver creates a new instance of per-open state in the open method and associates that instance with the new file descriptor by calling devfs_set_cdevpriv(9). This function accepts a void pointer argument and a destructor callback function. The destructor is invoked to clean the per-open state when the last reference to the file descriptor is closed. Other character device switch methods call devfs_get_cdevpriv(9) to retrieve the void pointer associated with the current file descriptor. Note that this family of functions always operates on the current file descriptor as determined implicitly by the caller context. The driver does not pass an explicit reference to a file descriptor to these functions.

In the first article in this series, we demonstrated support for per-instance data using the si_drv1 field.

Listing 3 shows the open and mmap_single methods as well as the **`cdevpriv`** destructor for a new **`memfd`** character device driver. This simple driver provides similar functionality to the `SHM_ANON` extension in FreeBSD's shm_open(2) implementation. Each open file descriptor of this device is associated with an anonymous VM object. The VM object's size grows, if necessary, when it is mapped. The VM object can be shared with other processes by sharing the file descriptor, for example by passing the file descriptor over a UNIX domain socket. To implement this, the driver allocates a new VM object in the open method and associates that VM object with the new file descriptor. The mmap_single object retrieves the VM object for the current file descriptor, grows it if necessary, and returns a reference to it. Finally, the destructor function drops the file descriptor's reference on the VM object.

Listing 3: Per-Open Anonymous Memory

```
static int
memfd_open(struct cdev *cdev, int fflag, int devtype, struct thread *td)
{
	vm_object_t obj;
	int error;

	/* Read-only and write-only opens make no sense. */
	if ((fflag & (FREAD | FWRITE)) != (FREAD | FWRITE))
		return (EINVAL);

	/*
	 * Create an anonymous VM object with an initial size of 0 for
	 * each open file descriptor.
	 */
	obj = vm_object_allocate_anon(0, NULL, td->td_ucred, 0);
	if (obj == NULL)
		return (ENOMEM);
	error = devfs_set_cdevpriv(obj, memfd_dtor);
	if (error != 0)
		vm_object_deallocate(obj);
	return (error);

}

static void
memfd_dtor(void *arg)
{
	vm_object_t obj = arg;

	vm_object_deallocate(obj);
}

static int
memfd_mmap_single(struct cdev *cdev, vm_ooffset_t *offset, vm_size_t size,
    struct vm_object **object, int nprot)
{
	vm_object_t obj;
	vm_pindex_t objsize;
	vm_ooffset_t delta;
	void *priv;
	int error;

	error = devfs_get_cdevpriv(&priv);
	if (error != 0)
		return (error);
	obj = priv;

	/* Grow object if necessary. */
	objsize = OFF_TO_IDX(round_page(*offset + size));
	VM_OBJECT_WLOCK(obj);
	if (objsize > obj->size) {
		delta = IDX_TO_OFF(objsize - obj->size);
		if (!swap_reserve_by_cred(delta, obj->cred)) {
			VM_OBJECT_WUNLOCK(obj);
			return (ENOMEM);
		}
		obj->size = objsize;
		obj->charge += delta;
	}

	vm_object_reference_locked(obj);
	VM_OBJECT_WUNLOCK(obj);
	*object = obj;
	return (0);
}
```

Extended Character Device Pagers

The mmap_single method mitigates some of the limitations of the default character device pager by permitting a character device to use VM objects backed by any pager as well as permitting a character device to associate different VM objects with different offsets. However, some limitations remain. The device pager is unique among other pagers in that it can map physical addresses that are not associated with physical RAM such as MMIO regions. Due to its use of unmanaged pages, there is no way to revoke mappings of the device pager nor a way for a driver to know if all mappings have been removed. FreeBSD 9.1 introduced a new interface to the device pager that provides solutions to both problems.

The new interface requires character device drivers to explicitly create device VM objects. These VM objects are then used by the mmap_single method to provide a backing store for mappings. In the new interface, the mmap character device method is replaced by a new method structure (`struct cdev_pager_ops`). This structure includes methods invoked when a VM object is created (`cdev_pg_ctor`), a page fault requests a page from a VM object (`cdev_pg_fault`), and a VM object is destroyed (`cdev_pg_dtor`). VM objects using the extended device pager are created by calling `cdev_pager_allocate()`. The first argument to this function is an opaque pointer stored in the handle member of the new VM object. This pointer is also passed as the first argument to the constructor and destructor pager methods. The second argument to `cdev_pager_allocate()` is the object type, either

`OBJT_DEVICE` or `OBJT_MGTDEVICE`. The third argument is a pointer to a **`struct cdev_pager_ops`** instance.

The **`cdev_pager_allocate()`** function only creates a single VM object for each opaque pointer. If the same opaque pointer is passed to a subsequent call to **`cdev_pager_allocate()`**, the function will return a pointer to the existing VM object instead of creating a new one. In this case, the VM object's reference count is increased, so **`cdev_pager_allocate()`** always returns a new reference to the returned VM object.

Let's make use of this interface to extend the original version of the mappage driver from Listing 1 so that it can be safely unloaded while there are no active mappings. In this case, we will use an `OBJT_DEVICE` VM object. This still uses unmanaged mappings of a single wired page allocated when the driver is loaded. However, there is now additional state needed to determine if that allocated page is in use, so this version of the driver defines a softc structure containing the pointer to the page, a boolean variable to track if the page is actively mapped, a boolean to track if the driver is being unloaded (in which case new mappings are disallowed), and a mutex to guard access to the boolean variables. A pointer to the softc structure is stored in the **`si_drv1`** field of the character device and is also used as the opaque handle for the VM object. The mmap_single character device method validates each mapping request (including failing requests while an unload is pending) and calls **`cdev_pager_allocate()`** to obtain a reference to the VM object mapping the wired page. Note that the mmap_single method doesn't have to handle the cases of creating a new VM object or reusing an existing VM object separately. The constructor pager method sets the boolean **`mapped`** softc member to true. Once the last mapping of the VM object is removed and the VM object is destroyed, the destructor pager method is called which sets the **`mapped`** softc member to false. The **`mappage_destroy()`** function fails to unload with the **`EBUSY`** error if the **`mapped`** member is true when an unload is requested.

The cdev_pager_allocate() function only creates a single VM object for each opaque pointer.

The page fault pager method is more complex than the mmap character device method it replaces. The page fault method works more directly with the VM system and how a fault is normally handled by VM pagers. When a page fault occurs, the VM system allocates a free page of RAM and invokes a pager method to fill that page with the appropriate contents. The swap and physical pagers zero new pages in this method, while the vnode pager reads the appropriate contents from the associated file. The default device pager takes a different route. Since it is generally designed to map non-RAM addresses such as MMIO regions, the default device pager allocates a "fake" VM page tied to the physical address returned by the mmap method and replaces the new VM page allocated by the VM system with the "fake" VM page (the new VM page is released back to the system as a free page). The page fault pager method allows a driver to implement either approach by passing in a pointer to the new VM page allocated by the VM system. The page fault pager method is responsible for either filling that page with suitable content or replacing it with a "fake" VM page. For our driver, we compute the physical address of our wired page the same as before but use that physical address to construct a "fake" VM page.

Listing 4 shows the mmap_single character device method, the three device pager methods, and the `mappage_destroy()` function called during module unload. In example 2, we suspend the `maprw` test program while it has the page from the mappage device mapped and attempt to unload the driver which fails. After resuming the test program and letting it unmap the device by exiting, the driver is unloaded successfully.

Listing 4: Using the Extended Device Pager

```
static struct cdev_pager_ops mappage_cdev_pager_ops = {
	.cdev_pg_ctor = mappage_pager_ctor,
	.cdev_pg_dtor = mappage_pager_dtor,
	.cdev_pg_fault = mappage_pager_fault,
};

static int
mappage_mmap_single(struct cdev *cdev, vm_ooffset_t *offset, vm_size_t size,
    struct vm_object **object, int nprot)
{
	struct mappage_softc *sc = cdev->si_drv1;
	vm_object_t obj;

	if (round_page(*offset + size) > PAGE_SIZE)
		return (EINVAL);

	mtx_lock(&sc->lock);
	if (sc->dying) {
		mtx_unlock(&sc->lock);
		return (ENXIO);
	}
	mtx_unlock(&sc->lock);

	obj = cdev_pager_allocate(sc, OBJT_DEVICE, &mappage_cdev_pager_ops,
	    OFF_TO_IDX(PAGE_SIZE), nprot, *offset, curthread->td_ucred);
	if (obj == NULL)
		return (ENXIO);

	/*
	 * If an unload started while we were allocating the VM
	 * object, dying will now be set and the unloading thread will
	 * be waiting in destroy_dev().  Just release the VM object
	 * and fail the mapping request.
	 */
	mtx_lock(&sc->lock);
	if (sc->dying) {
		mtx_unlock(&sc->lock);
		vm_object_deallocate(obj);
		return (ENXIO);
	}
	mtx_unlock(&sc->lock);

	*object = obj;
	return (0);
}

static int
mappage_pager_ctor(void *handle, vm_ooffset_t size, vm_prot_t prot,
    vm_ooffset_t foff, struct ucred *cred, u_short *color)
{
	struct mappage_softc *sc = handle;

	mtx_lock(&sc->lock);
	sc->mapped = true;
	mtx_unlock(&sc->lock);

	*color = 0;
	return (0);
}

static void
mappage_pager_dtor(void *handle)
{
	struct mappage_softc *sc = handle;

	mtx_lock(&sc->lock);
	sc->mapped = false;
	mtx_unlock(&sc->lock);
}

static int
mappage_pager_fault(vm_object_t object, vm_ooffset_t offset, int prot,
    vm_page_t *mres)
{
	struct mappage_softc *sc = object->handle;
	vm_page_t page;
	vm_paddr_t paddr;

	paddr = pmap_kextract((uintptr_t)sc->page + offset);

	/* See the end of old_dev_pager_fault in device_pager.c. */
	if (((*mres)->flags & PG_FICTITIOUS) != 0) {
		page = *mres;
		vm_page_updatefake(page, paddr, VM_MEMATTR_DEFAULT);
	} else {
		VM_OBJECT_WUNLOCK(object);
```

```
            page = vm_page_getfake(paddr, VM_MEMATTR_DEFAULT);
            VM_OBJECT_WLOCK(object);
            vm_page_replace(page, object, (*mres)->pindex, *mres);
            *mres = page;
      }
      vm_page_valid(page);
      return (VM_PAGER_OK);
}

...

static int
mappage_destroy(struct mappage_softc *sc)
{
      mtx_lock(&sc->lock);
      if (sc->mapped) {
            mtx_unlock(&sc->lock);
            return (EBUSY);
      }
      sc->dying = true;
      mtx_unlock(&sc->lock);

      destroy_dev(sc->dev);
      free(sc->page, M_MAPPAGE);
      mtx_destroy(&sc->lock);
      free(sc, M_MAPPAGE);
      return (0);
}
```

Example 2: Safely Unloading via the Extended Device Pager

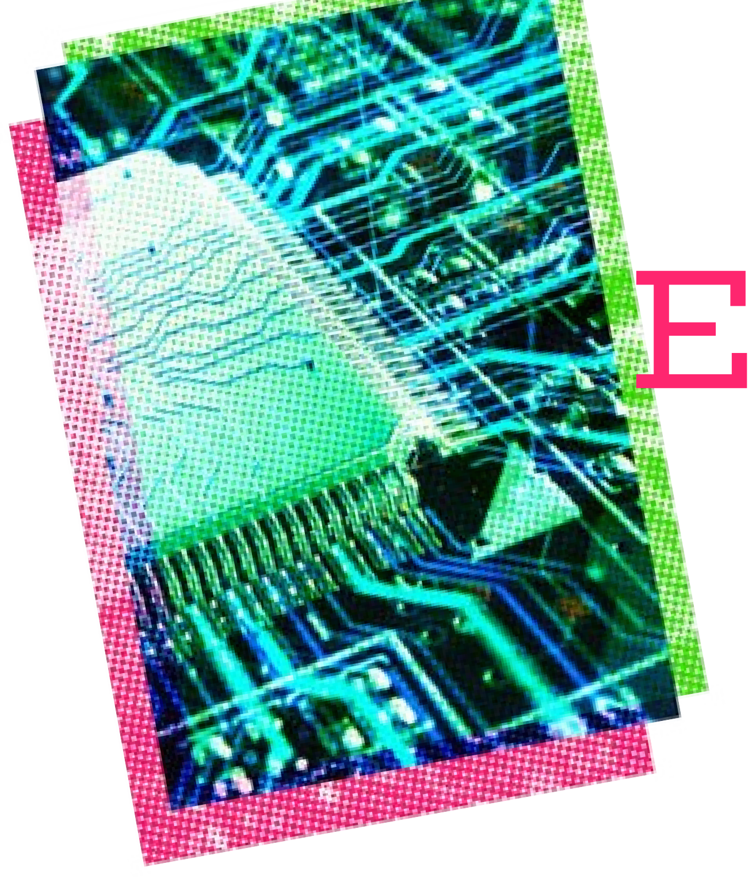
```
# maprw write /dev/mappage 16
^Z
Suspended
# kldunload mappage
kldunload: can't unload file: Device busy
# fg
maprw write /dev/mappage 16
maprw: empty read
# kldunload mappage
```

The extended device pager interface also adds a new type of device pager. The `OBJT_MGTDEVICE` pager differs from `OBJT_DEVICE` in that it always uses managed pages for mappings instead of unmanaged pages. This means that mappings for a page can be forcefully revoked even while the page is mapped. For fictitious pages mapping non-RAM pages, "fake" VM pages must be explicitly created before using them in the pager via the `vm_phys_fictitious_reg_range()` function.

Conclusion

In this article, we dove into some more unusual use cases for character devices including memory mappings and per-open state. Thanks for reading this series of articles. Hopefully, it was a useful introduction to character device drivers in FreeBSD.

JOHN BALDWIN is a systems software developer. He has directly committed changes to the FreeBSD operating system for over twenty years across various parts of the kernel (in-cluding x86 platform support, SMP, various device drivers, and the virtual memory subsys-tem) and userspace programs. In addition to writing code, John has served on the FreeBSD core and release engineering teams. He has also contributed to the GDB debugger. John lives in Ashland, Virginia with his wife, Kimberly, and three children: Janelle, Evan, and Bella.



Embedded FreeBSD: Learning to Walk–Interfacing to the GPIO System

BY CHRISTOPHER R. BOWMAN

In the last column, we created a simple circuit that blinked the LEDs on the board, and we learned two different ways to load this circuit into the FPGA. Sadly, when we loaded our circuit, the CPU stopped running. Furthermore, while this is mildly interesting, there is no interaction with the CPUs on the chip. In this column, we'll take a little more complex step into Vivado, learn how to keep our CPU running when we load circuits, and explore the GPIO system in FreeBSD.

Previously when we used either U-boot or `xbit2bin` and `/dev/devcfg` under FreeBSD we saw that FreeBSD halted. What I think happens is the processor's system stops running. Turns out the circuit `FPGA.bit` file we used didn't include configuration information for the processor system. In this installment, we'll fix this.

In this column, we'll take a little more complex step into Vivado.

I vacillated over how to present the information in this episode. From a learning standpoint, the most natural way to do this is probably using Vivado's GUI. On the other hand, GUIs do not lend themselves well to automation for the obvious reason that they require a human to run the GUI. Further, it is difficult and tedious to describe the GUI steps. Fortunately, Vivado has two features that make it relatively easy to work around this. When working with the GUI, the Vivado tool produces a `.jou` file which is a journal of all the TCL commands that the GUI is executing under the hood. Vivado also provides the TCL command `write_project_tcl` which can be used to recreate the project file that Vivado creates when you use the GUI. I generally prefer using the `.jou` file as I find the scripts more compact and understandable and if I run the scripts I can then either start the GUI or use `write_project_tcl` to write a project script. Scripts also seem a more natural fit for a revision control system like git.

If we look at "Figure 1-1: Zynq-7000 SoC Overview Figure 1-1: Zynq-7000 SoC Overview" from "UG585: Zynq-7000 SoC Technical Reference Manual" we can see that there are a variety of peripheral blocks (UART, I2C, SPI, etc.) which can be connected via a multiplexor to external pins. Section "1.2.3 I/O Peripherals" of the same manual details a bit more of the capabilities. For our purpose, now, we are just going to take note that we can route signals from the gpio device out to pins on the chip with a fair amount of flexibility. If we also look at the Arty Z7 Reference Manual section 12 "Basic I/O," we can see that the green LEDs on the board are connected to chip pins which sink to ground via current setting resistors. If we can set these pins high, the LEDs will light up, and conversely, if the pins are set low, the LEDs turn off.

To toggle these pins, we'll use the Vivado software to route the GPIO device outputs to the LED pins which will allow the GPIO device to control them. I didn't know it when I started this journey, but FreeBSD has a GPIO subsystem, and somebody has even kindly written a driver to make this all usable from user space.

FreeBSD has a GPIO subsystem, and somebody has even kindly written a driver to make this all usable from user space.

To get started, clone the git repo onto a Linux host with the Vivado tools (I showed how to set up a bhyve in a previous installment) and type **make** at the top of the repo. If you have the Vivado tools in your path and everything works right, it should run Vivado and pull in a script that will instantiate the processor subsystem and connect the first four EMIO pins of the GPIO device to the LED pins. The inclusion of the processor subsystem will fix the problem we previously had with the processors stopping when we programmed the device with a bit stream.

Look for the **zynq_gpio_leds.bit** file built by running **make** in the top level of the git repo. Program this into the chip as we did last time using the **xbit2bin** program:

```
# xbit2bin zynq_gpio_leds.bit
```

You should see exactly nothing happen. Not very exciting, but at least the processor should still be running.

Now, we need to use FreeBSD's GPIO subsystem. Typing **man gpioctl** gives a nice summary of what is possible.

As root, we can run the **gpioctl** program to list the available pins:

```
# gpioctl -f /dev/gpioc0 -l
```

Didn't work, did it? Yep, I was a little surprised by this, too. Looking at the GPIO source in **/usr/src/sys/arm/xilinx/zy7_gpio.c** I see there are probe and attach functions in the driver, but looking at my ARTYZ7 system **dmesg** output, I don't see anything indicating the device was found. Looking more closely at the probe function:

```
static int
zy7_gpio_probe(device_t dev)
{

        if (!ofw_bus_status_okay(dev))
                return (ENXIO);
        if (!ofw_bus_is_compatible(dev, "xlnx,zy7_gpio"))
                return (ENXIO);

        device_set_desc(dev, "Zynq-7000 GPIO driver");
        return (0);
}
```

I can see that just about the only thing required to find the device is having the function **`ofw_bus_is_compatible(dev, "xlnx,zy7_gpio")`** return true.

In embedded systems, like most ARM systems, the hardware generally isn't self-identifying like a modern PCIe bus. The software can't auto-identify what hardware is present and where its control registers are in the memory address space. For this reason, many operating systems use FDTs (Flattened Device Trees) to describe their device memory map. FDTs are text files that describe information about an embedded system including, among other things, what devices are present and where they are in memory. This allows the software to work with a variety of devices without needing to hard code information. The same kernel can often work with slightly different devices just by using a different FDT. FDTs are translated from DTS files (Device Tree Source) into DTB (Device Tree Binary) files via a tool called **`dtc`**, the device tree compiler. **`dtc`** has options that allow you to compile a DTS or decompile a DTB. The latter comes in very handy. For instance, you can ask the kernel for the DTB it's using and have **`dtc`** turn it into text with:

```
# sysctl -b hw.fdt.dtb | dtc -I dtb -O dts
```

If we look for the gpio section, we see (among other things) this:

```
          gpio@e000a000 {
               compatible = "xlnx,zy7_gpio";
             };
```

The compatible string in the DTB isn't what the driver is expecting, so it assumes the device isn't present. If you look at the FreeBSD boot output, you'll see that the kernel is using the DTB from the EFI firmware emulation that U-boot is providing. We could change this by fixing what U-boot is providing, but that would require patching and recompiling u-boot. Instead, FreeBSD provides the capability to load a DTB file at the kernel loader prompt or using a **`loader.conf`** variable. You can load a DTB file from the loader using the following **`/boot/loader.con`**f variables:

```
# fdt_name="/boot/dtb/zynq-artyz7.dtb"
# fdt_type="dtb"
# fdt_load="YES"
```

This works but there is also a third way. Turns out you can create FDT/DTS overlays which are patches to an FDT/DTS/DTB. We just need a DTS overlay that adds the correct compatible string and we should be good to go. Here is the heart of the DTS overlay I've included in the **`dts`** directory with a **`Makefile`** from the project repo:

```
&{/axi/gpio@e000a000} { compatible = "xlnx,zy7_gpio"; };
```

We may look at FDT files in more detail in a future column, so I won't explain it here. Instead, I'll just give you a flavor of the file. Once you've built the DTB overlay, take the generated DTB and put it in **/boot/dtb/overlays**, and add the following to **/boot/loader.conf**:

```
fdt_overlays="artyz7_gpio_overlay.dtb"
```

Reboot and note the new **dmesg** output:

```
gpio0: <Zynq-7000 GPIO driver> mem 0xe000a000-0xe000afff irq 5 on simplebus0
```

Now let's try that **gpioctl** command again and you should see a line like this among others:

```
pin 64:   0 EMIO_0<IN>
```

We need to tell the GPIO subsystem to configure the pin as output, and then we can try toggling it:

```
gpioctl -f /dev/gpioc0 -c EMIO_0 OUT
gpioctl -f /dev/gpioc0 -t EMIO_0
```

I'll wait. Try not to stand up and dance when the light comes on. See if you can figure out how to turn on the other LEDs and then run the script in **scripts/blink.sh** with an argument of 2. You should see the LEDs blink as they count in binary.

If you've got questions, comments, feedback, or flames on any of this I'd love to hear from you. You can contact me at articles@ChrisBowman.com.

CHRISTOPHER R. BOWMAN first used BSD back in 1989 on a VAX 11/785 while working 2 floors below ground level at the Johns Hopkins University Applied Physics Laboratory. He later used FreeBSD in the mid 90's to design his first 2 Micron CMOS chip at the University of Maryland. He's been a FreeBSD user ever since and is interested in hardware design and the software that drives it. He has worked in the semiconductor design automation industry for the last 20 years.

Contents

The Handling of SYN Segments in FreeBSD

BY RANDALL STEWART AND MICHAEL TÜXEN

TCP Connection Setup

The Transmission Control Protocol (TCP) is a connection-oriented transport protocol providing a reliable bidirectional byte stream service. The TCP connection setup requires three TCP segments to be exchanged, which is called a three-way handshake. The TCP endpoint initiating the TCP connection and sending the first TCP segment, the SYN segment, is called the client. The TCP endpoint waiting for the first TCP segment is called the server and responds to the received SYN segment with a SYN ACK segment. When the client receives this SYN ACK segment, it completes the handshake by sending an ACK segment.

The TCP handshake is not only used to synchronize the state between the two endpoints including the initial sequence numbers for providing reliability, but also to negotiate the use of TCP extensions via TCP options. With today's Internet, the most widely deployed TCP options during the handshake (contained in the SYN and SYN ACK segments) are:

1. The maximum segment size (MSS) option
   The MSS option contains a 16-bit number (between 0 and 65535), which is the maximum number of payload bytes the sender of this option is willing to receive in a single TCP segment. For this number, it is assumed that no options at the IP layer and the TCP layer are used. In case such options are used, the number must be decremented by the size of the options. This helps the TCP sender to avoid sending TCP segments requiring fragmentation at the IP layer.
2. The SACK-permitted option
   This option announces that the sender can handle selective acknowledgments (SACK options). This improves the performance in case of packet loss.
3. The TCP Window Scale option
   This option contains a natural number between 0 and 14. If both sides send this option, the receive window scaling is enabled. This allows the use of a larger receive window than would be allowed by the format of the TCP header, where the receive window is limited to 16 bits (and therefore 65535 bytes). This avoids the fact that the size of the receiver window field in the TCP limits the throughput of a TCP connection.
4. The TCP Timestamp option
   This option contains two 32-bit numbers, which often encode some timing information in millisecond granularity. It is used to improve the TCP performance.

TCP is specified using a state event machine. Initially, an endpoint is in the CLOSED state. When the endpoint is willing to accept TCP connections (on the server side), the TCP endpoint is moved to the LISTEN state. When receiving the SYN segment from the client side and replying with a SYN ACK segment, the endpoint enters the SYN RECEIVED state. Once the TCP endpoint receives the ACK segment sent by the client, the TCP endpoint enters the ESTABLISHED state. These states can be observed using the `netstat` or `sockstat` command line tools.

The Application Programmers Interface (API) used to control TCP endpoints is the socket API. Programs normally use a listening socket, which tells the TCP implementation that it is OK to accept TCP connections on this endpoint, and for each accepted TCP connection the programs use a separate socket for each TCP connection. Applications can set parameters on the listening socket and most of the time these settings are then inherited by the accepted sockets. This article focuses on the TCP connection setup on the server side. It should be noted that this functionality applies to all TCP stacks (default, RACK, BBR, …).

This article focuses on the TCP connection setup on the server side.

SYN Flooding Attacks

When TCP was initially implemented, a new TCP endpoint was created whenever a SYN segment was received for a TCP endpoint in the LISTEN state. This required a memory allocation and resulted in a new TCP endpoint in the SYN RECEIVED state. All necessary information including the information related to the TCP options received in the SYN segment was stored in the TCP endpoint. This was done without any verification of the information provided, the IP address, and the TCP port number.

This allowed an attacker to send many SYN segments to a server and the server would allocate TCP endpoints until it ran out of resources. Therefore, the attacker could perform a denial-of-service attack, since once the server has no resources available anymore, it would not accept SYN segments from valid clients. The attacker only needs to send SYN segments, in particular, the attacker would not respond to any SYN ACK segments received. The attacker can even use spoofed IP addresses (IP addresses the attacker does not own).

The goal of this SYN flooding attack is that the receiver runs out of resources and is therefore not able to provide the service it is intended to provide. In FreeBSD, there are two mitigations implemented in the TCP stack for dealing with SYN flooding attacks:

1. Reducing the amount of memory allocated when a TCP endpoint moves from CLOSED to SYN RECEIVED state. This is done by using the SYN cache described in the next section.
2. Not allocating any memory when processing the incoming SYN segment. This is done by using SYN cookies as described in the section following SYN cache.

SYN Cache

The initial implementation of the SYN cache was added to the FreeBSD source tree in November 2001. It reduces the memory overhead of TCP endpoints in the SYN RECEIVED state by not allocating a full TCP endpoint, but a TCP SYN cache entry (`struct syncache` as defined in `sys/netinet/tcp_syncache.h`) instead. A TCP SYN cache entry is smaller than a TCP endpoint and only allows storage of information relevant in the SYN RECEIVED state. This information includes:

- The local and remote IP address and TCP port number.
- The information relevant for performing timer-based retransmissions of the SYN ACK segment.
- The local and remote TCP initial sequence number.
- The MSS reported by the peer in the MSS option of the received SYN segment.
- The local and remote window scaling shift values exchanged in the relevant window scaling options of the SYN and SYN ACK segments.
- Whether window scaling, timestamp, and SACK support were negotiated.
- Accurate ECN state.
- Additional IP layer information.

When a SYN segment is received for a listening endpoint, a SYN cache entry is allocated, the relevant information is stored in it and a SYN ACK segment is sent in response. If SYN cookies are disabled and there is a bucket overflow, the oldest SYN cache entry in the bucket is tossed. If the corresponding ACK segment is received, a full TCP endpoint is created with the data from the SYN cache entry and then the SYN cache entry is freed. The SYN cache also assures that the SYN ACK segment is retransmitted in case the corresponding ACK segment is not received in time.

When a SYN segment is received for a listening endpoint, a SYN cache entry is allocated.

The `sysctl`-variable `net.inet.tcp.syncookies` (which defaults to 1) control whether SYN cookies, as described in the next section, will be used in combination with the SYN cache to cover the case where no SYN cache entry can be allocated or looked up.

The SYN cache is vnet specific and organized as a hash table. The number of buckets is controlled by the loader tunable `net.inet.tcp.syncache.hashsize` (default 512). The maximum number of SYN cache entries in each hash bucket is controlled by the loader tunable `net.inet.tcp.syncache.bucketlimit` (default 30). There is also an overall limit of SYN cache entries given by the loader tunable `net.inet.tcp.syncache.cachelimit` (default is 15360 = 512 * 30). The number of currently used SYN cache entries is reported by the read-only `sysctl`-variable `net.inet.tcp.syncache.count`.

There are additional `sysctl`-variables relevant to the SYN cache. These are:

- `net.inet.tcp.syncache.rst_on_sock_fail`
  Controls for sending an RST segment or not in case a socket can't be created successfully (which defaults to 1).
- `net.inet.tcp.syncache.rexmtlimit`
  The maximum number of retransmissions of a SYN ACK segment (which defaults to 3).
- `net.inet.tcp.syncache.see_other`
  Control the visibility of the SYN cache entries (which defaults to 0).

The TCP SYN cache allows the server side to perform a fully functional handshake with a minimized memory resource. There is no functional difference to using a full TCP endpoint for TCP endpoints in the SYN RECEIVED state. Even tools like `netstat` or `sockstat` will report the entries from the SYN cache.

Supporting additional TCP options is not a problem, since the TCP SYN cache entry can be expanded.

The `sysctl`-variable `net.inet.tcp.syncookies_only` (defaulting to 0) can be used to disable the use of the SYN cache. In this case, only SYN cookies described in the next section will be used.

SYN Cookies

An additional level of protection against SYN flooding attacks was added to the SYN cache implementation in December 2001. Instead of allocating a smaller amount of memory when processing a received SYN segment, the relevant information is stored in a so-called SYN cookie and sent to the client in the SYN ACK segment. Then the client is expected to reflect the SYN cookie in the ACK segment. When the ACK segment is finally processed by the server, all relevant information is in the SYN cookie and the ACK segment. So, the server can create a TCP endpoint in the ESTABLISHED state. This way a SYN flooding attack does not result in any memory exhaustion. However, the generation of the SYN cookie mustn't require too many CPU cycles. If the SYN cookie generation is not cheap CPU-wise, it might allow a denial-of-service attack: this time not against the memory resource, but against the CPU resource.

The MAC uses a secret key, which is updated every 15 seconds.

The only field in the TCP header that can be chosen arbitrarily by the server and is reflected by the client is the initial sequence number of the server. This field is a 32-bit integer and therefore used as the SYN cookie.

In FreeBSD, these 32 bits are split into a 24-bit message authentication code (MAC) and 8 bits, which are used as follows:

- 3 bits for encoding one of 8 MSS values: 216, 536, 1200, 1360, 1400, 1440, 1452, 1460. For MSS values sent by the client in the MSS option not in this list, the largest value not exceeding the given one is used.
- 3 bits for encoding if the peer does not support window scaling or uses one of the 7 values: 0, 1, 2, 4, 6, 7, 8. If the client was sending a value not in the list, the largest value not exceeding the given one is used.
- 1 bit for encoding whether the client sent the SACK-permitted option or not.
- 1 bit for selecting one of two keys.

The MAC uses a secret key, which is updated every 15 seconds. The current and the last secret keys are kept around and used based on the bit in the SYN cookie for selecting the secret key.

The computation of the MAC includes the local and remote IP addresses, the initial sequence number of the client, the above 8 bits, and some internal information. From the MAC, 24 bits are generated and combined with the 8 bits from above, the SYN cookie is constructed.

When the ACK segment of the three-way handshake is received by the server, the MAC is verified. If that is successful, the TCP endpoint is created based on the information in the SYN cookie, which provides an approximation of the MSS option, an approximation of the window shift, and whether the client announced support for the SACK extension. All the other relevant information must be recovered from the ACK segment. This recovered information includes the local and remote IP addresses and port numbers, the local and remote initial sequence numbers whether the TCP timestamp option is used or not, and in the case it is, what the current parameters are.

Comparison of SYN Cache and SYN Cookies

The advantage of SYN cookies compared to the SYN cache is very clear: no memory allocation when a new SYN ACK segment is received. However, using SYN cookies also has its downsides:

- The MSS is approximated by 8 values, all smaller than or equal to 1460. Therefore, there is no support for MTUs larger than 1500 bytes for IPv4.
- The shift used for window scaling is approximated by 7 values, all smaller than or equal to 8. This means that larger window shifts above 8 are not supported and thus the connection will have a smaller window size.
- No support for TCP options other than the ones widely deployed right now. This makes it hard to support new TCP options used to negotiate new TCP features.
- No retransmissions of the SYN ACK segment, if a SYN ACK segment is lost the endpoint initiating the connection will have to retry sending its SYN segment.
- No visibility of TCP endpoints in the SYN RECEIVED state.
- No support for IP-level information.

Using the SYN cache does not have any of these limitations and is transparent but requires a memory allocation for each TCP endpoint in the SYN RECEIVED state.

Combined Usage of SYN Cache and SYN Cookies

Using only SYN cookies provides a better mitigation against SYN flooding attacks than using the SYN cache, but it comes with the drawback of limited functionality. Therefore, the default configuration of FreeBSD enables the SYN cache in combination with SYN cookies. This means that when a received SYN segment is processed, a SYN cache entry is generated, and the SYN ACK segment being sent contains a SYN cookie. If a SYN cache bucket overflows, it is assumed that this happens due to an ongoing SYN flooding attack, and therefore using the SYN cache is paused. During this time, only SYN cookies are used.

This additional functionality, introduced in September of 2019, gives the advantages of the SYN cache during normal operation, but also the improved protection of SYN cookies when a SYN flooding attack is going on.

RANDALL STEWART (rrs@freebsd.org) has been an operating system developer for over 40 years and a FreeBSD developer since 2006. He specializes in Transports including TCP and SCTP but has also been known to poke into other areas of the operating system. He is currently an independent consultant.

MICHAEL TÜXEN (tuexen@freebsd.org) is a professor at the Münster University of Applied Sciences, a part-time contractor for Netflix, and a FreeBSD source committer since 2009. His focus is on transport protocols like SCTP and TCP, their standardization at the IETF and their implementation in FreeBSD.

Contents

Fall 2024 FreeBSD Summit

BY ALICE SOWERBY

I attended the Fall 2024 FreeBSD Summit at the NetApp San Jose Campus in Santana Row. The event gave me a great opportunity to connect with the FreeBSD community, exchange ideas, and discuss ongoing projects. My main goal was to highlight the role of directly funded projects in the work of the FreeBSD Foundation through by giving a talk as well as through informal conversation.

Meeting people face-to-face reinforced how valuable in-person engagement is. I had many useful discussions about the impact of Foundation-funded initiatives, and I saw a lot of interest in FreeBSD's usability for laptops, which was central to my talk.

During the event, I met several interesting people who shared their experiences with, and ambitions for, FreeBSD. During the same trip, I was also glad to have the opportunity to meet the team at Framework which has helped me to understand more about the context of the Foundation's laptop project.

I delivered my talk, *Learning as We Grow: Managing FreeBSD Infrastructure and Laptop Projects at The FreeBSD Foundation*, on Friday, Nov 8. The audience asked thoughtful questions, and I appreciated the chance to share insights into how we structure and fund these projects.

One talk that especially stayed in my thoughts was *The History of the BSD Daemon* by Marshall McKusick, which provided interesting insights into the history of BSD. He also brought along t-shirts from past BSD conferences and gave them away, which was quite a touching moment for all involved.

Santana Row provided a lively and convenient setting for the summit. The NetApp building felt spacious and welcoming, which made it easy to meet people and have meaningful discussions. The event was well organized, and the setting encouraged productive conversations.

Since the summit, I have already seen the benefits of the connections I made and the conversations I had. The discussions and insights I gained have helped shape ongoing projects and reinforced the importance of community-driven development efforts.

Attending the Fall 2024 FreeBSD Summit gave me a chance to engage with the FreeBSD community, share knowledge, and strengthen collaborations. The event helped highlight the value of the FreeBSD Foundation's investments and provided a great forum for discussions on FreeBSD's future. I look forward to continuing these conversations and building on the momentum from the summit.

ALICE SOWERBY has a wealth of experience working to build teams and develop leaders in the tech sector.

Contents

Conference Report

FOSDEM 2025

BY TOM JONES

There are a lot of ways to be involved with Open Source software. Development is the obvious thing that will jump to everyone first, but many people aren't developers and there are other ways to contribute.

You can write code, but you can also test in-progress developments and releases, read and correct documentation, write new documentation, answer questions in one of a thousand places, work on advocacy, raise money, coordinate contributors, or run events. Running events is an amazing way to give back to the community, but it can take an incredible amount of work just to get ten people together.

We are lucky in the BSD world that we have three major geographically diverse conferences, and any given year the calendar is bursting with events. Ask anyone trying to slot in a hackathon about the number of conflicts they must work around.

We are lucky in the BSD world that we have three major geographically diverse conferences.

Most Open Source projects don't have the required scale and infrastructure to run a regular conference. Meeting is difficult, and some people may be lucky enough to get help from their employer to meet other developers a couple of times a year. The cost required to put on a conference is too much for most small projects and interaction is often online only.

In Europe, FOSDEM fills an important ecosystem niche. FOSDEM is the largest Open Source conference in the world and is open to all. The event completely takes over the UCB campus in Brussels and boasts 8,000 or so attendees. FOSDEM handles the difficult logistics of finding a venue and gives projects a place to meet, rooms to discuss and present, and places to show off demos. Such a large event doesn't just enable developers of projects to meet up, it also creates a lot of cross-pollination between projects possible.

To keep everyone busy, there is a main track of talks, hundreds of projects, or topic-specific dev rooms, and stands for projects to show their presence. FOSDEM needs projects of all sizes to run, and in exchange, it gives those attendees access to everyone else who comes to Brussels.

Selling FreeBSD at FOSDEM 2025

I volunteered to help staff the FreeBSD table at FOSDEM. In the lead-up to the event, a large envelope of stickers and a table runner arrived at my house, and I helped to coordinate the delivery of a banner to Belgium.

To help projects communicate about what they do, FOSDEM offers tables or booths. These act as a static place for users new and old to come and ask questions and give projects the chance of random passers-by stumbling onto an interesting project.

I spent most of the two days standing behind the table answering questions, giving away stickers, and discussing FreeBSD ideas with other project members.

Staffing the table at an Open Source Project is a great way to see how much you know. At times, there was a near-constant stream of people dropping by, ranging from current users and past users to those completely new to the project. Here are some of the questions I was asked during FOSDEM about how we advocate for FreeBSD.

- What desktop does FreeBSD run?
- What can you do with FreeBSD?
- Can I have a sticker?
- Who uses FreeBSD?
- Why pick FreeBSD over << my favourite >> Linux distro?
- Convince my friend here to use FreeBSD.
- Can I run containers on FreeBSD?
- How is your project funded?

What are your answers?

I found these questions and the others—especially the ones in French I didn't understand and completely missed—an excellent touch point for the mind of the Open Source community.

Hearing the questions people answer in this environment is telling for what is of interest to people thinking about other Operating Systems (or those poor confused people thinking we were a Linux distro after five no's).

Staffing the table at an Open Source Project is a great way to see how much you know.

The focus on switching and desktops shows a big gap in how we are presenting the project when we attend events. Stickers are wonderful and banners help people find us, but we lacked some easy demos that show—yes, FreeBSD is a desktop and can do almost anything you can do on Linux. The desktop environment questions were interesting, many of the questioners had already siloed us into a bucket with small, focused Linux distros built around a single desktop environment. We can run them all, and while satisfying for us, wasn't a compelling enough response to stick with anyone visiting the table.

these questions, I came away full of ideas for how we can better sell FreeBSD--with desktop use cases, but also the cool, compelling features we have. It wouldn't take much to bring along some SBCs and show the same release FreeBSD running on them all—no config needed.

More Advocacy

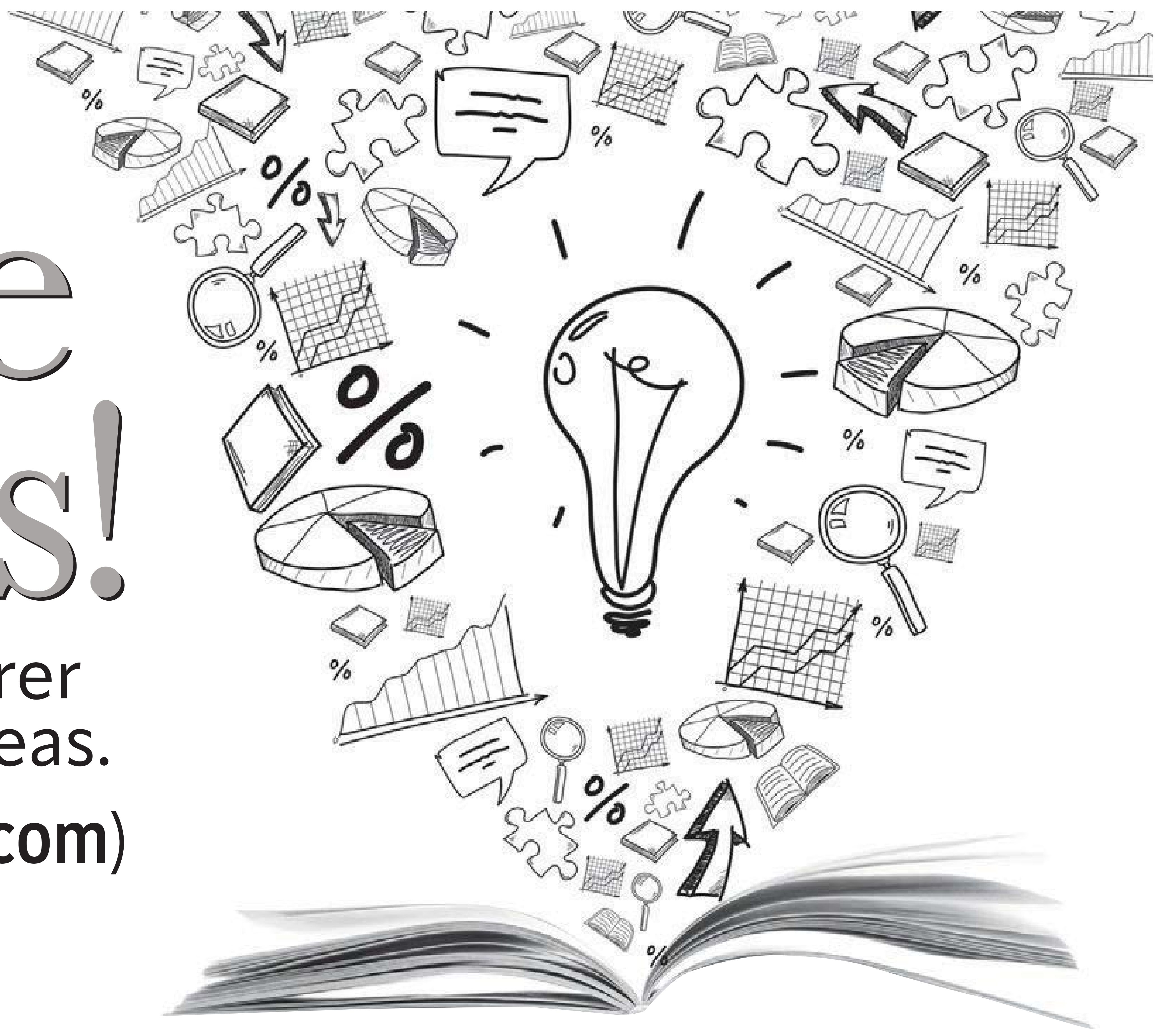
Look back at the questions. Do you have your answers ready?

If you have been to a conference, or community event or seen a charity's table in a mall, you know the setting. What can we do to be better able to answer questions like these?

The table at an Open Source conference is a high-contact place where you get hours of endless opportunities to interest people in FreeBSD. We came with stickers, mugs, and cheat sheets, but the most success is going to come from lasting positive experiences we can give people in these situations.

If you have an idea of how to better advocate for FreeBSD at conferences, please send me an email (thj@freebsd.org) and we can start talking about selling FreeBSD to the Open Source community.

TOM JONES is a FreeBSD committer interested in keeping the network stack fast.



Contents

2025 Events Calendar

BSD Events taking place through September 2025

BY ANNE DICKISON

Please send details of any FreeBSD related events or events that are of interest for FreeBSD users which are not listed here to freebsd-doc@FreeBSD.org.

June 2025 FreeBSD Developer Summit

June 11-12, 2025
Ottawa, Canada
https://wiki.freebsd.org/DevSummit/202506

Join us for the June 2025 FreeBSD Developer Summit, co-located with BSDCan 2025, which will take place in Ottawa, Canada. The two-day event takes place June 11-12, 2025, consisting of developer discussion sessions, vendor talks, and working groups.

BSDCan 2025

June 11-14, 2025
Ottawa, Canada
https://www.bsdcan.org/2025/

BSDCan is a technical conference for people working on and with BSD operating systems and related projects. It is a developers conference with a strong focus on emerging technologies, research projects, and works in progress. It also features Userland infrastructure projects and invites contributions from both free software developers and those from commercial vendors.

EuroBSDCon 2025

September 25-28, 2025
Zagreb, Croatia
https://2025.eurobsdcon.org/

This yearly conference gives the exceptional opportunity to learn about the latest news from the BSD world, witness contemporary deployment case studies, and personally meet other users and companies using BSD-oriented technologies. EuroBSDCon is also a boiler plate for ideas, discussions, and information exchange, which often turn into programming projects.

Contents