



®

FreeBSD®

July/August/Septmeber 2025

JOURNAL

Topic: Embedded

**Starting Firewall Development:
Tom Jones Interviews Igor Ostapenko**

CHERI^{IoT}

FreeBSD, Home Assistant, and rtl_433

Writing Effective Bug Reports

**Implementing a Quantum-Safe Website
On FreeBSD**

**Getting Started with WiFi Development,
Part 2, Working on a Driver**



FreeBSD[®] JOURNAL

The FreeBSD Journal is Now Free!

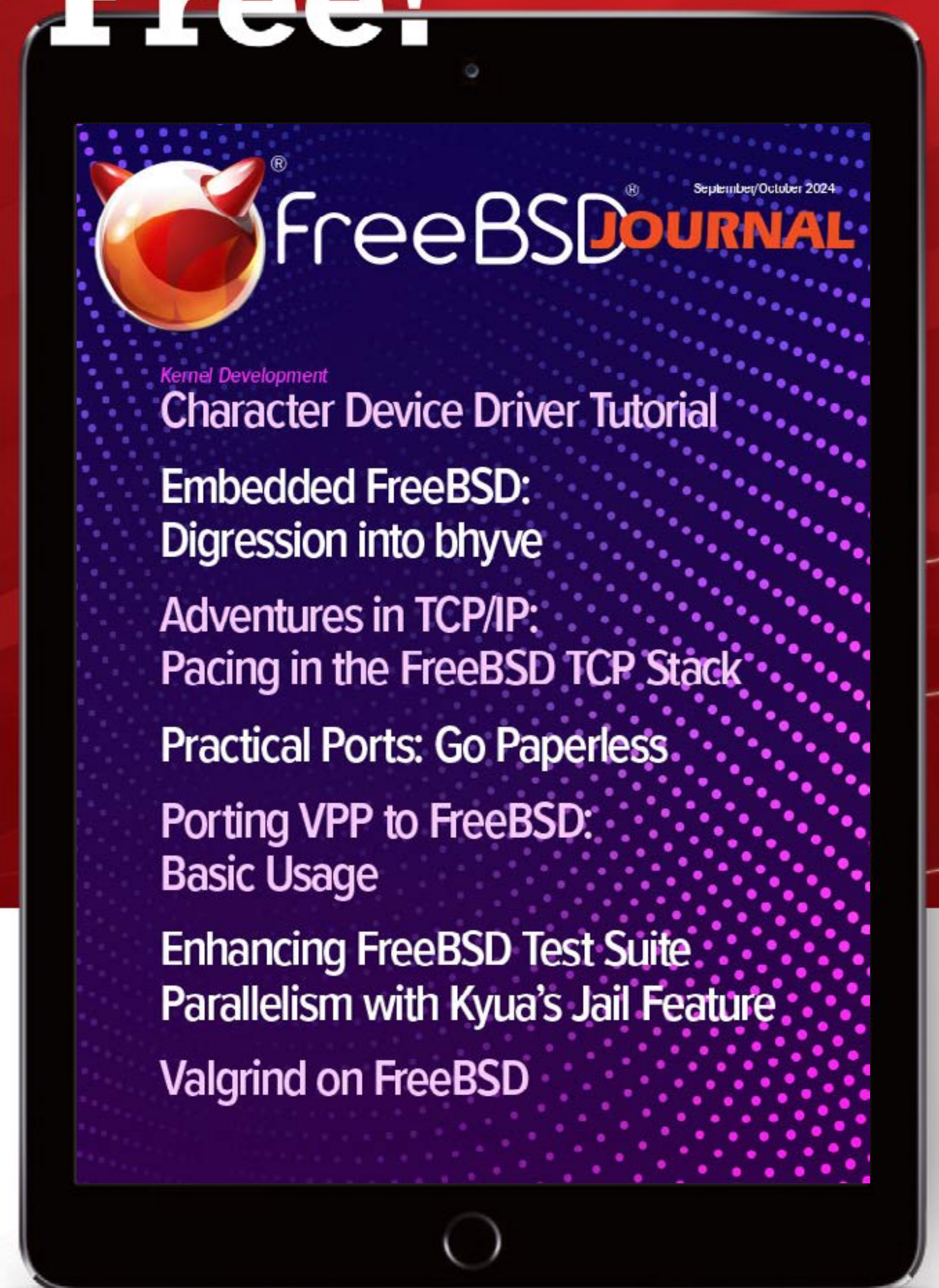
Yep, that's right Free.

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!

2025 Editorial Calendar

- Jan/Feb/March Downstreams
- April/May/June Networking
- July/August/Sept Embedded
- Oct/Nov/Dec FreeBSD 15.0



Find out more at: freebsd.foundation/journal

Editorial Board

John Baldwin • FreeBSD Developer and Chair of the *FreeBSD Journal* Editorial Board

Tom Jones • FreeBSD Developer, Software Engineer, FreeBSD Foundation

Ed Maste • Senior Director of Technology, FreeBSD Foundation and Member of the FreeBSD Sec Team

Benedict Reuschling • FreeBSD Documentation Committer

Jason Tubnor • BSD Advocate, Senior Security Lead at Latrobe Community Health Service (NFP/NGO), Victoria, Australia

Mariusz Zaborski • FreeBSD Developer

Advisory Board

Anne Dickison • Deputy Director, FreeBSD Foundation

Justin Gibbs • Founder of the FreeBSD Foundation, President of the FreeBSD Foundation Board, and a Software Engineer at Facebook

Allan Jude • CTO at Klara Inc., the global FreeBSD Professional Services and Support company

Dru Lavigne • Author of *BSD Hacks* and *The Best of FreeBSD Basics*

Michael W Lucas • Author of more than 40 books including *Absolute FreeBSD*, the *FreeBSD Mastery* series, and *git commit murder*

Kirk McKusick • Lead author of *The Design and Implementation* book series

George Neville-Neil • Past President of the FreeBSD Foundation Board and co-author of the *Design and Implementation of the FreeBSD Operating System*

Hiroki Sato • Chair of AsianBSDCon, Member of the FreeBSD Core Team, and Assistant Professor at Tokyo Institute of Technology

Robert N. M. Watson • Director of the FreeBSD Foundation Board, Founder of the TrustedBSD Project, and University Senior Lecturer at the University of Cambridge

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 4 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@freebsd.foundation.org

Copyright © 2025 by FreeBSD Foundation. All rights reserved.
This magazine may not be reproduced in whole or in part without written permission from the publisher.

LETTER

from the Foundation

Welcome to the third *FreeBSD Journal* issue of 2025! Summer has ended in the northern hemisphere, along with the resulting change in weather in many places. In the FreeBSD project, we are racing towards the finish line for 15.0 with the release scheduled near the end of the year. Our next issue will focus on 15.0, highlighting several of the new features and changes in 15.0.

This issue includes articles spanning a range of topics. Chris Bowman continues his excellent article series on using FreeBSD with the Arty Z7. Christos Margiolis describes his experience at BSDCan 2025 and the associated FreeBSD developer summit. David Chisnall introduces CHERIoT, an application of the CHERI architecture to 32-bit RISC-V. Vanja Cvelbar narrates his adventure of using SDR with FreeBSD as part of a home automation system, and Gergely Poór explores the use of quantum-safe cryptography on FreeBSD. Lastly, Tom Jones is quite busy in this issue, interviewing Igor Ostapenko, continuing his Wi-Fi development series, and providing sage advice on submitting bug reports.

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

John Baldwin

Chair of the *FreeBSD Journal* Editorial Board



Topic: Embedded

8 Starting Firewall Development: Tom Jones Interviews Igor Ostapenko

By Tom Jones

12 CHERIoT

By David Chisnall

20 FreeBSD, Home Assistant, and rtl_433

By Vanja Cvelbar

28 Writing Effective Bug Reports

20 By Tom Jones

34 Implementing a Quantum-Safe Website On FreeBSD

By Gergely Poór

43 Getting Started with WiFi Development, Part 2, Working on a Driver

By Tom Jones

3 Foundation Letter

By John Baldwin

5 We Get Letters

By Michael W. Lucas

53 Embedded FreeBSD: Looking Back and Forward

By Christopher R. Bowman

57 Conference Report: BSDCan 2025

By Christos Margiolis

60 Events Calendar

By Anne Dickison

WeGetletters

by Michael W Lucas



Greetings, Letters Columnist!

Before you treat my letter like those IRS notices and fling it into /dev/null, please note that — if I timed this correctly — a delivery driver will be knocking on your door any minute now with a gallon of handmade gelato, all in the hopes of evoking a useful answer from you.

What does your average sysadmin need to know about embedded systems?

—Mostly Expecting Derision, Sadly

Dear BAD,

A valiant effort. Your email was delayed by greylisting, however, so it arrived after the delivery truck. I have no idea where you found pickled salmon and durian gelato, but I commend the effort you put into achieving a new height of appalling. Well done. You're a natural sysadmin.

What does a sysadmin need to know about embedded systems? An embedded system is just a computer. What makes embedded systems different from regular hosts? Absolutely nothing, except you must do everything correctly.

You can tune a regular host. When /var/log/ on your web server consumes an inordinate amount of disk, you can adjust the log rotation conditions or tweak what you log. System tuning in ossified enterprises with rigorous change control processes enforced by an authoritarian goon whose soul was crushed by discovering he was born too late to offer his services as an unpaid intern to Gaius Julius Caesar Germanicus is lax compared to changing proper embedded systems. Changing an embedded system requires a system update, and do you know how often these devices get reflashed? That's right, *never*. Maybe you let your streaming media device reboot when it downloads the newest firmware, but when was the last time you updated your fridge? I avoid this class of problem by walking into the appliance store and saying, "You can sell me anything so long as it includes a complete lack of Internet connectivity." They promptly lead me to

**What does a sysadmin need to know about embedded systems?
An embedded system is just a computer.**

the 20th Century Room. I leave with a botnet-resistant dishwasher. Expensive, but worth it. If you're even asking this question, you have chosen poorly.

An embedded system configured less than perfectly will stop working. Maybe that's "stop working correctly," maybe it starts again once you restart it, maybe it full-on bricks itself. Whatever. It's fixable, of course. Many embedded systems are designed for repair by replacement. A few have terminals where you can check for full disks or wedged processes but hide it. Disk drives all have firmware, and many manufacturers offer arcane tricks to access the terminal. Run a few wires from a DB9 to select pins on a disk drive, type the secret code, and poof — you have hard-wired access directly into the drive's feeble little brain. There's even a menu system. Once you exclude the treachery, the menu consists primarily of lies, but it exists.

Everything is embedded systems. All the way down.

Your RAID controller? It runs an embedded operating system that lets it emulate a wholly different RAID controller from the 1980s because sysadmins insist that they be allowed to set a "stripe size" that doesn't map to anything inside the actual storage medium. Hit F1 or DELETE during boot, and you'll get into the mainboard's embedded system. The "BIOS Log" is full, by the way. It's been full for years. Nothing useful has been logged since your predecessor was hired. Everything in your computer is a separate embedded system supporting your non-embedded system. Your "bare metal" server isn't bare metal, because "bare metal" does not exist.

**Everything is
embedded systems.
All the way down.**

I've griped about virtualization many times, but everything is virtual and the whole world merits that same scorn.

Why put an operating system in your hard drive, mainboard, or USB chipset? Hardware changes are expensive and require talking to the manufacturer. Software changes are inexpensive and can be jammed into existing hardware. We weasel around hardware bugs with differently buggy software. Yes, releasing correct hardware would be better, but the people in charge assure us that's not possible, so reality once again demonstrates that what seemed like a good idea was actually a horrible idea. The embedded system saves the manufacturer money but increases pain.

Don't give me that look. All computers increase pain.

The purpose of a system is what it does, and a computer does pain.

We cope with the pain by adding more computers.

The first Rule of System Administration declares "computers were a mistake," and examining the mental health impact of the teetering cataclysm of systems inside your computer proves it, let alone the systems in your car. Don't get me started on your car. Requiring everyone in the United States to borrow, purchase, or otherwise "acquire" a multi-ton kinetic energy weapon was bad enough, but add in separate computers for the brakes and the accelerator and the climate control, and congratulations! You've built a Rolling Debacle. When your drive-by-wire steering catches a SIGABRT the only question is who

gets debacled, you or bystanders. I would say “innocent bystanders,” but nowadays everybody uses a computer, so nobody is innocent. Even folks who want to remain unsoiled are compelled to use a website or an app because the guilty make themselves feel better by dragging everyone down to their sewer. (“Come to my party! It’ll be great! Never mind the floaters!”)

Can you even find a computer without a gratuitous embedded system somewhere? Certainly. It’s called an abacus. My abacus has a series of five-bit bytes and a separate register with two-bit bytes. Quite sophisticated. The operating system is implemented in hand position. Best of all, nobody has written a TCP/IP stack for the abacus.

The bad news is that embedded systems have become more accessible than ever. Inexpensive, low-power systems are available from your least abhorred retailer, enabling you to build tiny systems to control your whole home if you wish. If you want a custom remote control for the big door on the shed where you store your multi-ton kinetic energy weapon, you can do so for the low cost of a hundred bucks of hardware, several man-months of effort, and your will to live.

So, what should you know about embedded systems? If you’re stuck with one and it breaks, pretend you’re an idiot and call tech support. Otherwise, you’ll delve into the system and discover just what atrocities the designer committed in the name of system stability. That won’t go anywhere pleasant.

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




MICHAEL W LUCAS is the author of *Absolute FreeBSD*, *Dear Abyss*, *SSH Mastery*, and more. The new edition of *Networking for System Administrators* has recently escaped. <https://mwl.io>

Books that will help you. Or not.

“While we appreciate Mr Lucas’ unique contributions to the Journal, we do feel his specific talents are not being fully utilized. Please buy his books, his hours, autographed photos, whatever, so that he is otherwise engaged.”

— John Baldwin
FreeBSD Journal Editorial Board Chair

<https://mwl.io>



Starting Firewall Development

AN INTERVIEW WITH IGOR OSTAPENKO
BY TOM JONES

TJ: *There are numerous paths into working on FreeBSD, some through university courses and others through work experience. How did you learn about the project, and what initially drew you to Operating System development?*

IO: In the 1990s, during my school years, I had the chance to work with several programming languages and technologies, e.g., I spent hours tinkering with TR-DOS, carefully planning line numbers (which triggered flashbacks when I started using ipfw) and crafting DATAs and GOTOs for yet another game I envisioned, as well as experimenting with interrupt vectors for resident programs in MS-DOS. Because of that, a minimalistic command-line interface was not new to me. So, when I encountered FreeBSD during my Computer Science studies at university, my only question was which books to get.

It was clear that mastering FreeBSD would be challenging but incredibly rewarding, as I'd have to acquire some fundamental knowledge along the way. Why FreeBSD? The seniors promoted FreeBSD because our entire dorm network was built on it. This was the early 2000s, still carrying the inertia of the previous decade, when FreeBSD was the de facto standard for networking. In recent years, I've been focusing more on operating system development. With a broad background in software development, I've built a collection of ideas for how OS internals could be leveraged for high-level solutions.

“

When I encountered FreeBSD during my Computer Science studies at university, my only question was which books to get.

”

TJ: *What were your steps to making your first FreeBSD changes? How did you decide what to work on initially?*

IO: The first steps were about preparation. I wanted to refine my existing knowledge of FreeBSD, fill gaps, and develop a more structured vision. A well-known resource for this is *The Design and Implementation of the FreeBSD Operating System* book by Marshall Kirk McKusick, George V. Neville-Neil, and Robert N.M. Watson.

The FreeBSD source code was not entirely new to me; I already had a general sense of its structure formed over the years. However, I wanted a professional guide to ensure I did not miss key concepts, stylistic nuances, or structural elements. The world is lucky to have the FreeBSD Kernel Internals: An Intensive Code Walkthrough course by Marshall Kirk McK-

usick. It saved a lot of my time, covered all my questions, and provided valuable historical context to address the “Why?” questions in the best possible way. Additionally, the FreeBSD Networking from the Bottom Up course by George V. Neville-Neil offered further refinement of the networking stack side.

I considered the pros and cons of starting with big projects versus smaller ones and discussed it with mckusick@ and kib@. Konstantin Belousov recommended starting with a smaller task, such as bug fixing, which proved to be the most effective approach. I started working on the most recent pf bugs reported, which spawned other related improvements in the jail subsystem, test tools like Kyua’s `execenv=jail` project, and even a new module, `dummysmbuf`, for specific network tests. As a result, I continue to contribute to improving the project, in collaboration with Kristof Provost, Mark Johnston, and other FreeBSD developers.

TJ: *Bugs are a great way for a newcomer to start in any project. Do you have any recommended areas for new FreeBSD contributors to begin in 2025?*

IO: With official guidance and specific directions for new contributors available on the project website, e.g., <https://wiki.freebsd.org/IdeasPage>, I would like to consider an alternative approach. There are many possible paths to contributing, and the best ones are likely those that align with personal interests.

For example, if there is an interest in learning and using a FreeBSD network tool or kernel module such as `netstat`, `route`, `pf`, `ipfw`, `netgraph`, etc., then working with respective documentation and manual pages might open opportunities to improve them by providing extra examples, reworking complex concepts, or adding missing parts. If such a tool or module is not present in FreeBSD, then adding useful programs to the FreeBSD Ports Collection or keeping them up to date is another vital way to help. Such projects are usually fun and educational, as they may require a deeper understanding of FreeBSD kernel interfaces to succeed.

If the goal is a deeper understanding of the kernel code, then a similar formula could be applied — choosing the topic that is in use or is planned to be used could provide more benefits. For instance, if it’s a firewall, then a better understanding of how its rules work behind the scenes gives power users an edge. It could involve investigating the routing mechanisms if there is an interest in finding a solution for a non-generic problem. There might be a need to implement missing features or RFCs in the kernel. There is periodic interest in porting existing solutions from other platforms, such as Netlink implementation in FreeBSD, with the respective work-in-progress migration of the existing tools or Vector Packet Processing (VPP) framework porting, with open doors for further enhancements. Sooner or later, working with the existing code reveals opportunities for optimizations — spending less resources, such as time, per unit of data transferred or processed, benefits all businesses relying on FreeBSD.

“

If the contribution is something more than a small patch, I recommend two initial steps: homework and communication.

”

If the contribution is something more than a small patch, I recommend two initial steps: homework and communication. There are many ways to reach FreeBSD developers (see <https://www.freebsd.org/community/>), and the basic one is the mailing lists like hackers@FreeBSD.org. It's a good idea to discuss a potential project first to find a general agreement on the direction or to discover if someone is already working on it, as is usual for open-source projects; the more homework done, the better the communication outcome.

TJ: *The development process is quite daunting, and there are many dead ends. Are there any shortcuts you would like to share with new developers to make debugging and development easier?*

IO: I believe the *FreeBSD Journal* is an excellent source of shared professional experience. I recommend scanning the TOCs of the previous issues to find topics that fill gaps or offer a new perspective on a familiar subject. For instance, new developers may be interested in articles such as "Kernel Development Recipes" and "DeBUGGING the FreeBSD Kernel" by Mark Johnston [*FreeBSD Journal*, Sep/Oct 2021][*FreeBSD Journal*, Nov/Dec 2018], "FreeBSD Kernel Development Workflow" by Navdeep Parhar [*FreeBSD Journal*, Mar/Apr 2024], and "More Modern Kernel Debugging Tools," which you wrote [*FreeBSD Journal*, Mar/Apr 2024]. These and similar articles provide a quick overview of build system capabilities and shortcuts (such as avoiding lengthy rebuilds), or how virtualization and other 3rd-party software could be leveraged to improve productivity. Each article is not a complete book that covers all details or mentions all available options, but they do provide good starting points.

To begin forming good practices encouraged within the project, I recommend checking the article "Writing Good FreeBSD Commit Messages" by Ed Maste [*FreeBSD Journal*, Sep/Oct 2020]. Sooner or later, it's a good idea to become acquainted with the article "FreeBSD Code Review with git-arc" by John Baldwin [*FreeBSD Journal*, Sep/Oct 2021]. This tool significantly improves the process of patch publishing, reviewing, updating, and landing.

In terms of work on the networking part of the kernel, I suggest taking a closer look at FreeBSD's Jails and VNET features. If the work does not involve specific hardware support or other non-generic topics, then VNET-based

Jails can be leveraged to ease the testing of a new networking feature during development. To interest a new developer, VNET-based Jails can be roughly treated to build a network of lightweight virtual machines to test specific packet paths or network stack behavior. This approach can be easier than other methods, even if the same packet buffer (mbuf) may end up playing all the roles in the scene, since nothing leaves the host. Another important outcome of this exploration is that such a playground may become a good starting point for developing a new automated test for a newly implemented feature or a fixed bug. This leads to the article "The Automated Testing Framework" by Kristof Provost [*FreeBSD Journal*, Mar/Apr 2019], which introduces the FreeBSD Test Suite. The source of existing firewall tests based on VNET Jails (src/tests/sys/netpfil) and Kristof's talks [<https://www.youtube.com/watch?v=gTyt7KLz1mw>] about that can be used for inspiration.

Nevertheless, I would invest time in setting up work with the code. The kernel is an unusual type of software that supports multiple architectures, compilers, and exceptional cases, where the preprocessor's ifdefs may not be enough, and some conditions are resolved outside of the code itself. In addition, the source code includes some generated code, such as the boilerplate parts of system calls or VFS operations. Hence, grepping or using an

editor/IDE with default settings is not sufficient. This is a subjective topic, and my “Neovim + clangd + intercept-build” setup works fine for me, so it’s better to get an overview of available options:

<https://docs.freebsd.org/en/articles/freebsd-src-lsp/>

Using Language Server Protocol greatly simplifies code comprehension and navigation, which is essential for a new developer.

TJ: *Thank you for your time for this interview. Is there any final advice to new contributors or anything you would like to add?*

IO: Thanks for your time as well. The general advice applies to any international open-source project with a considerable volunteer base: maintain an open-minded attitude and some level of patience. Let me add a networking-based allegory here. There are many other hosts (contributors) in the community network, and they are not available at any time using any protocol we prefer. Each host is likely busy with its workload, and its network interfaces may already be overloaded. So, a metaphorical connection timeout towards us doesn’t necessarily mean a strict RST; more likely, it means that more time is needed to process our SYN. Communication approaches like email can provide deep buffering, which may be handled as first-in, last-considered, so a polite retransmission is often a practical step. Latency expectations also need reconsideration due to time synchronization issues (time zones), priorities (work over volunteering), or downtimes (weekends, vacations, etc.).

This network has been running for decades, with some hosts participating since the early days. Their accumulated knowledge and experience can help us with new challenges, guide us in case of dilemmas, or keep us away from issues not yet visible to us. At the same time, newly joined hosts may bring fresh ideas or new visions to consider. Each contributor brings something unique to complete or extend the whole puzzle. Hence, staying open-minded and striving to understand the intent or idea behind a message received should provide new contributors with better connectivity.

Although not mandatory or expected, leaning towards the symmetrical bandwidth of your host improves overall network capacity and fosters its expansion. In other words, be prepared to open some ports to accept incoming connections on your side.

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

IGOR OSTAPENKO is a contributor to FreeBSD and OpenZFS with extensive software development experience across various domains, including systems for manipulating and testing navigation devices, enterprise solutions for optimizing business processes, reverse-engineering, and B2B/B2C startups.

CHERIot

BY DAVID CHISNALL

The [CHERI](#) project has always had a close relationship with FreeBSD. It began from observing that Capsicum-based compartmentalization was great for new code but retrofitting it to existing libraries (with one process per library instance) was difficult for two reasons:

First, libraries want to share complex data structures, which imposes a lot of serialization overhead when turning the interfaces into messages sent over some inter-procedural communication (IPC) channel. A function call in a normal library would simply share a data structure by passing a pointer to an object. A privilege-separated library would need to authorize everything moved between the caller and callee. Libraries also often want long-term sharing, which imposes additional synchronization overhead.

Second, processes are isolated using a memory management unit (MMU), which provides a virtual-memory abstraction with mappings from addresses in a virtual address space to the underlying physical memory. Modern MMUs are fast because they have a translation look-aside buffer (TLB), a fast cache of translations. The TLB caches virtual to physical address translations. If a single page is shared between ten processes, it will take ten TLB entries. MMUs are great for isolation but poor for sharing.

These two problems led to the general observation: Isolation is easy, sharing is hard.

[CHERI](#) is a set of architectural extensions that provide fine-grained memory safety for everything from assembly code on up. CHERI, like Capsicum, is a capability system. In a capability system, every action must be accompanied by a capability, an unforgeable token of authority, that authorizes the action.

In contrast, a lot of other optimized mechanisms have a notion of *ambient authority*: you can perform an action simply because you are you. For example, if a process runs with UID 0 on a traditional UNIX system (without something like the FreeBSD MAC framework or SELinux), it can do a lot of privileged operations, whether it intended them or not.

Processes running as root are often security problems because it's easy to trick them into doing things that they shouldn't. In a capability system, a privileged process would instead hold a set of tokens that each authorize specific actions. It would need to choose the one to use at each point, preventing accidental use of elevated privileges.

CHERI is a set of architectural extensions that provide fine-grained memory safety for everything from assembly code on up.

MMUs have a similar notion of ambient authority. A running program can access any memory for which there is a valid mapping. If you have a buffer overflow, the MMU doesn't care that you didn't mean to access an adjacent object: you have the right to access that memory and so you can. The relevant piece of code may also hold a pointer to the other object, and so be authorized to access that object, but it didn't mean to at that point.

Capsicum extends file descriptors to be capabilities.

A Capsicum file descriptor has a rich set of permissions and, after entering capability mode (via the **cap_enter** system call) a process cannot access anything outside of its own memory without providing a valid file descriptor (capability), with the correct permissions, to a system call.

For example, in a normal POSIX process, you can call **open** to access any file that the user (or some MAC policy involving the user, the process, and maybe other things) has access to. In contrast, a capability-mode process must call **openat** and pass it a capability that authorizes access to a specific directory. This enforces the *principle of intentionality* and avoids a large class of vulnerabilities.

For example, if you mean to access something in a temporary directory, but happen to be given a path to some more important directory that you should not be accessing (for example, `../etc/rc.conf`), **openat** with a file descriptor to your temporary directory will fail (correctly, preventing an exploit) but **open** would succeed.

CHERI protects memory accesses in a way analogous to how Capsicum protects filesystem accesses. A conventional instruction set architecture has load, store, and jump instructions (sometimes, as with x86, combined with more complicated operations) that take an address as a base. This provides a model of memory that's similar to conventional UNIX's view of the filesystem: Any load or store can work if the process is allowed to load or store at that location. Buffer overflows or use-after-free bugs are to memory what path-traversal vulnerabilities are to the filesystem.

In a CHERI system, this address is replaced by a CHERI capability, an unforgeable value that authorizes access to a range of the address space. These values are stored in registers and memory and are protected by the hardware from tampering.

How does this make it easy to program?

Does a programmer have to track capabilities as some additional thing?

No, not at all.

It turns out that most programming languages already have an abstraction for a token that authorizes you to access a region of memory. They call it a pointer (or, in some cases, a reference). As a programmer, targeting a CHERI system, you mostly don't think about CHERI capabilities at all, you simply think about pointers.

If you do some arithmetic that takes a pointer out of the range of the object, you can't use it for loads or stores anymore. And, because the hardware knows precisely which things in memory are pointers and which aren't, it is possible to invalidate pointers when objects are freed. This means that you can share objects with a library on a CHERI system simply by passing pointers to those objects as arguments to a function exposed from the library.

Buffer overflows or use-after-free bugs are to memory what path-traversal vulnerabilities are to the filesystem.

The very earliest operating system for CHERI was a tiny microkernel but the vast majority of it was done on FreeBSD. CheriABI (a CHERI userspace ABI for FreeBSD) demonstrated a complete memory-safe userspace and a kernel on a friendly fork of FreeBSD ([CheriBSD](#)), which aims to upstream the changes before the 16.x release. A CHERI base architecture is currently being standardized as part of RISC-V, and FreeBSD 16 should have first-class support for all of the upcoming application-class CHERI cores implementing this instruction set.

FreeBSD was essential to the CHERI project. CHERI was a long-term hardware-software co-design project, which required modifying both hardware and software parts of the stack to explore where and how various ideas could best be implemented. This required a production-grade operating system that was easy to modify. The clear structure and well-defined abstractions of FreeBSD made this easy. We've seen with later Linux-on-CHERI work that the effort of adapting FreeBSD was far lower, and the project would probably not have had enough software engineers to have done the same on Linux as an initial target. The permissive license also made it easy to show vendors of other operating systems how various hardware features were used. FreeBSD was also an early adopter of LLVM, which has similar advantages in terms of ease of modification and license. It's easy to compile the entire FreeBSD base system with a modified LLVM, which makes testing new CPU features trivial. Brooks Davis wrote a much longer [article](#) about the benefits of FreeBSD for CHERI research in the May / June 2023 edition of the *FreeBSD Journal*.

In 2019, a few of us at Microsoft decided to see whether we could scale down the same abstractions to the smallest systems.

Scaling down CHERI

CheriABI demonstrated that you could run real POSIX applications, including large programs such as Chromium, on a full KDE desktop with Wayland and 3D drivers, in a memory-safe world. This could coexist with existing binaries, via a COMPAT64 layer, which worked much like the COMPAT32 layer that allows FreeBSD to run 32-bit programs on 64-bit systems. Most systems from mobile phones up to servers use the set of abstractions that were shown to work.

In 2019, a few of us at Microsoft decided to see whether we could scale down the same abstractions to the smallest systems.

We had three questions:

- Can you make the things that work well with CHERI on 64-bit systems work on 32-bit ones?
- If you have CHERI, what can you discard?
- What does an operating system look like if you can assume CHERI from the ground up?

The first is a slightly non-obvious problem. The abstractions that CHERI provides don't seem to be dependent on the size of an address, except for one thing: all of the metadata for a CHERI capability must fit in the same number of bits as the address. This means that on a 32-bit system we have half as much space for metadata.

CHERI uses a compressed encoding for the bounds that takes advantage of the fact that there is a lot of redundancy between the base of an object, the top of an object, and the address of a pointer to that object. On a smaller system, there is less redundancy and

less space for the bounds. Fortunately, on embedded systems, the total amount of address space tends to be smaller, so there's less need to be able to precisely represent very large regions of the address space. A high-end microcontroller typically has well under 4 MiB of total RAM, so most objects are very small.

We also had fewer bits available for permissions than on a 64-bit system and had to compress our permission encoding, eliminating combinations that were either bad for security, not useful, or not meaningful.

How I used FreeBSD in developing CHERIoT

When we built the first [CHERIoT](#) prototypes, there were three core components:

- A CPU core written in BlueSpec SystemVerilog.
- A port of LLVM.
- A clean-slate RTOS.

BlueSpec SystemVerilog is a Haskell-based high-level hardware description language, which makes rapid prototyping easy. The [BlueSpec compiler](#) required a few small tweaks to build on FreeBSD, which we upstreamed, and FreeBSD is now a supported platform. It isn't yet in ports, but that would be great to see. With this, we could build a simulation of the core that ran on FreeBSD. Later, we moved to a production-quality code implemented in SystemVerilog and used verilator (from packages) to build a simulator on FreeBSD.

Like FreeBSD, CHERIoT RTOS is a permissively licensed, community-developed, operating system.

LLVM development on FreeBSD is very easy.

FreeBSD is a first-class target for LLVM and LLVM is trivial to build. In some ways, it was too easy: when we needed to support people on LTS releases of some Linux distributions, we found that bootstrapping the version of LLVM that we were using was hard because it used a newer version of C++ than their stock toolchain supported.

For cutting-edge work, FreeBSD was far easier: multiple versions of GCC and LLVM were available in ports. This also made it easy to reproduce some of the build failures on other systems, by simply installing the old version of GCC that they shipped and configuring a build to try using it. Once those were working, cross-compiling and testing the RTOS was easy.

Lessons learned from FreeBSD

Like FreeBSD, CHERIoT RTOS is a permissively licensed, community-developed, operating system.

Most importantly, CHERIoT aims to copy FreeBSD's model of designing features before implementing them. The easiest time to change code is before it's written. This is how FreeBSD ended up with features like Jails, kqueue, and Capsicum: careful thought and design iteration, rather than throwing an API at users, hoping it works, and then living with the consequences.

We learn from Capsicum that we can make things that look to programmers like conventional file descriptors or handles into capabilities. Our software abstractions follow this model.

We also learned from **kqueue** that a single, simple, unified way of polling for any blocking event is easy. The **kqueue** design does not map particularly well to a privilege-separated RTOS, but the core idea does. Our scheduler exposes futexes (a simple atomic compare-and-wait-if-equal operation) as the only blocking event source. Interrupts are mapped to futexes, so a thread waits for an interrupt by simply waiting on a futex. The scheduler then layers a multiwaiter API on top, allowing a thread to wait for multiple events, either from hardware or software.

Perhaps most importantly, we've learned from FreeBSD that documentation is king. It doesn't matter how amazing your system is if no one can figure out how to use it. Between well-written man pages and the Handbook, FreeBSD is easy to learn. Writing a book for developers wanting to pick up CHERIoT RTOS was a priority and it was published earlier this year. On top of that, we have doc comments for every API, which are parsed by modern IDEs (and vim with our version of **clangd** as the language-server protocol implementation).

Lessons FreeBSD could learn from CHERIoT

CHERIoT RTOS is currently written entirely in C++. C++ has a lot of advantages over C for systems programming. It's easy to create rich abstractions that are checked at compile time. For example, we have a message queue design that uses a single counter for each of the producer and consumer pointers and has to handle the cases where these values wrap.

In C++, we can define **constexpr** functions to do the increments, and then write templates that **static_assert** over their behavior. Every time that we compile the file that defines these, the compiler will exhaustively check the overflow behavior on every possible value of producer and consumer pointers for some small queue sizes.

Using rich types also lets us avoid a lot of errors by construction. For example, we have a **Permission-Set** class that manages the set of permissions on a CHERIoT capability. This is a **constexpr** set that lets you construct permissions by name and will generate the bitmap that the instructions that operate on permissions expect.

Using rich types
also lets us avoid a lot
of errors by construction.

In the loader, we use this to describe both the permissions that we want on a capability and the permissions that each of the roots have. We will get a compile failure if we try to derive a permission that is not present in the original, which is far easier to debug than a later instruction failing because it lacks an expected permission in one of its operands.

We make a lot of use of a pattern where we have an inline templated function that does some compile-time checks and is optimized away, and a calls a type-erased function. Our type-safe logging works like this, for example. We have a **printf**-like function that takes an array of the arguments to print and the union discriminators. This is constructed by some templates that generate the discriminator values based on the types, so whether you want to log a pointer, an enumeration value, or a MAC address, you'll get the correct output, without having to correctly match the type in the string and with compile-time checks.

This is extensible. MAC addresses, for example, are not built in, the network stack defines a callback for them and a template specialization that handles the mapping.

We don't yet have a Rust compiler (it's coming soon!) but when we do, we expect to start using Rust for some components as well. Richer types in systems programming languages let us both avoid bugs at compile time and also write a lot less code.

In a lot of key places in the RTOS, we'd need much more source code if we used C, and that code would be harder to maintain. It would also be far harder to find developers who are familiar with the language. Today, the number of lines of new code that we see written for systems languages are ordered:

1. C++
2. C
3. Rust

It's easier today to find C++ developers than either of the other two, but that ignores the trend. C++ has been seeing fairly slow growth since C++11 was introduced. C has been seeing a much sharper steady decline in the same time. Rust was seeing steady growth, but it's accelerated over the last three years. I expect that it will be easier to find Rust developers than C developers very soon and probably easier to find Rust developers than C++ within a decade or so.

This is unsurprising.

Developers tend to favor languages in which they can be more efficient. There is ongoing work in FreeBSD to support Rust in the base system but there's a much simpler path to adopting modern C++, which can make it far simpler to express complex concepts than C and provide a lot more compile-time checks.

FreeBSD has also done some fantastic work using Lua for things that are not absolutely performance critical, including userspace parts, policies in the bootloader, and ZFS channel programs. Lua is a far simpler language to learn than any of the above list and is easy for relatively inexperienced programmers to be productive in. We use Lua in our build system (and for typesetting the CHERIoT book!), but sadly the Lua VM takes about as much memory as a typical microcontroller has in total, so we can't use it in the RTOS.

Richer systems programming languages are important but FreeBSD's use of Lua is a good reminder that a lot of things—even in the kernel—don't actually need a systems programming language.

Developing for CHERIoT on FreeBSD

The CHERIoT toolchain is in ports on FreeBSD, as is the **xmake** build tools, and so you can install them simply with:

```
# pkg ins cheriot-llvm xmake-io git
```

That gives you the prerequisites for building CHERIoT firmware. Note that, at the time of writing, the version of CHERIoT LLVM in the quarterly branch is 18, whereas the version in the latest branch is 20 (and about to be updated to 21), so it's a good idea to use the latest branch for development.

You should now be able to try cloning the RTOS and building a simple firmware image.

First, clone the RTOS:

```
$ git clone --recurse https://github.com/CHERIoT-Platform/cheriot-rtos
$ cd cheriot-rtos
```


Next, you need to configure the build:

```
$ cd examples/01.hello_world/
$ xmake config --sdk=/usr/local/llvm-cheriot
checking for platform ... cheriot
checking for architecture ... cheriot
Board file saved as build/cheriot/cheriot/release/hello_world.board.json
Remapping priority of thread 1 from 1 to 0
generating /tmp/cheriot-rtos/sdk/firmware.rcode.ldscript.in ... ok
generating /tmp/cheriot-rtos/sdk/firmware.ldscript.in ... ok
generating /tmp/cheriot-rtos/sdk/firmware.rwdata.ldscript.in ... ok
$ xmake
...
[100%]: build ok, spent 13.796s
```

If you have a lowRISC Sonata board, you can add **--board=sonata** to the end of the **xmake config** line and you'll get an ELF file targeting their board. The board shows up as a USB mass-storage device with a FAT filesystem that can load a UF2 file. If you do **xmake run**, it will point you to the missing Python package that you need to install from **pip** to convert the ELF to a UF2 file. Once this is installed, it will either tell you which file to copy, or copy it with no further interaction if the **SONATA** filesystem is mounted in a common location.

If you don't, then you can run the resulting firmware in a simulator. By default, these examples will target the Sail simulator. This requires OCaml and does build on FreeBSD, but requires some manual steps. The project also ships a Linux dev container, which works very nicely in the FreeBSD Linuxulator with Podman:

```
# pkg ins podman-suite
# podman run --rm -it --os linux -v path/to/cheriot-rtos:/home/cheriot/cheriot-rtos
ghcr.io/cheriot-platform/devcontainer:x86_64-latest
```

This will drop you into an ephemeral container with your clone of the RTOS source code mounted in **/home/cheriot/cheriot-rtos**. Note that the current version of Podman in ports doesn't like using the tag that refers to a multi-arch container when the OS doesn't match the host. We provide binaries for x86-64 and AArch64, so if you're on AArch64 simply replace **x86_64** with **aarch64** in the above line.

The dev container has all of the tools installed in **/cheriot-tools**, so you can try building the example again in the same way as earlier:

```
$ cd examples/01.hello_world/
$ xmake f --sdk=/cheriot-tools
...
$ xmake
...
[100%]: build ok, spent 13.524s
$ xmake run
Board file saved as build/cheriot/cheriot/release/hello_world.board.json
Remapping priority of thread 1 from 1 to 0
Running file hello_world.
ELF Entry @ 0x80000000
tohost located at 0x80006448
```



```
Hello world compartment: Hello world
SUCCESS
```

Congratulations, you've run memory-safe C++ code in a simulator built from a formal model of the ISA!

The dev container contains three other simulators:

- The cycle-accurate verilator simulator for the Ibex core with a minimal set of peripherals.
- The simulator for lowRISC Sonata boards.
- The MPact simulator from Google that provides a high-performance simulator with an integrated debugger

The MPact simulator is compatible with the Sail (simplified) machine.

You can try running it directly:

```
$ /cheriot-tools/bin/mpact_cheriot build/cheriot/cheriot/release/hello_world
Starting simulation
Hello world compartment: Hello world
Simulation halted: exit 0
Simulation done: 106508 instructions in 0.2 sec (0.5 MIPS)
Exporting counters
```

You can do the development in FreeBSD and run just the simulators in the container, or do all of the development in the container. On other platforms, most developers use an editor such as Visual Studio Code that has dev container integration. This should also be possible on FreeBSD, configuring Podman to run the Linux version of the container.

Alternatively, you can build all of the simulators natively for FreeBSD. The instructions for this are too long for this article, but look at the documentation in the RTOS repository for guidance. All of the dependencies for them are in ports and hopefully the simulators themselves will be soon!

DAVID CHISNALL's background spans operating systems, compilers, hardware, and security. He is the author of the Definitive Guide to the Xen Hypervisor, has been an LLVM committer since 2008 and was a member of the FreeBSD Core Team from 2012 to 2016. He joined the CHERI project to lead the compilers and languages thread of the research at the University of Cambridge in 2012. He continued to work on CHERI, including leading the creation of CHERIoT, at Microsoft from 2018 to 2023. He is co-founder and Director of Systems Architecture at SCL Semiconductor, which makes CHERIoT SoCs, and co-maintainer of the CHERIoT Platform open-source project.

FreeBSD, Home Assistant, and rtl_433

BY VANJA CVELBAR

A long time ago, I started playing with Home Assistant, and I'm very proud of a few automations. There is a motion sensor in the bathroom that turns on the light. If the sun has already set, the light turns on with a very dim orange light; otherwise, it uses the maximum power and white color. There is an automation that turns on the dehumidifiers if the relative air humidity exceeds a predefined value. Additionally, there is a safety measure. Unfortunately, we have a cellar window that is prone to flooding if the rain comes from the south, which is a rare occurrence. I have installed an external and an internal moisture sensor; when triggered, we get a message on Telegram and can act accordingly. The whole system is hosted on a bhyve virtual machine running HAOS.

The need

With two radio-controlled thermometers on the balcony and in the garden, I sought a simple overview and, more importantly, a clear graph of the outside temperature. Those are simple units periodically sending their data over the radio on 433MHz. After researching the topic, it was clear that a standard Software Defined Radio adapter could listen to the signals and process them further. Combining it with the great rtl_433 and rtl-sdr projects was the answer.

After connecting an RTL 2832-based dongle to my laptop, I started getting data from my thermometers, but also from the neighbors and from the TPMS systems of the cars passing by.

The nice part was that the rtl_433 software already supports posting data to a MQTT server. There was already one running on Home Assistant for the ZigBee network, so that involved just creating a new user to post to it.

```
/var/log/rtl_433.log
```

```
{"time" : "2025-04-13 18:42:55.702554", "protocol" : 19, "model" : "Nexus-TH", "id" : 209, "channel" : 1, "battery_ok" : 1, "temperature_C" : 13.200, "humidity" : 61, "mod" : "ASK", "freq" : 433.910, "rssi" : -0.431, "snr" : 30.360, "noise" : -30.791}
{"time" : "2025-04-13 18:42:56.539182", "protocol" : 8, "model" : "LaCrosse-TX", "id" : 31, "temperature_C" : 11.300, "mic" : "PARITY", "mod" : "ASK", "freq" : 433.892, "rssi" : -12.208, "snr" : 19.251, "noise" : -31.459}
```

JSON log of rtl_433 decoding the external thermometers radio transmissions.

The first implementation

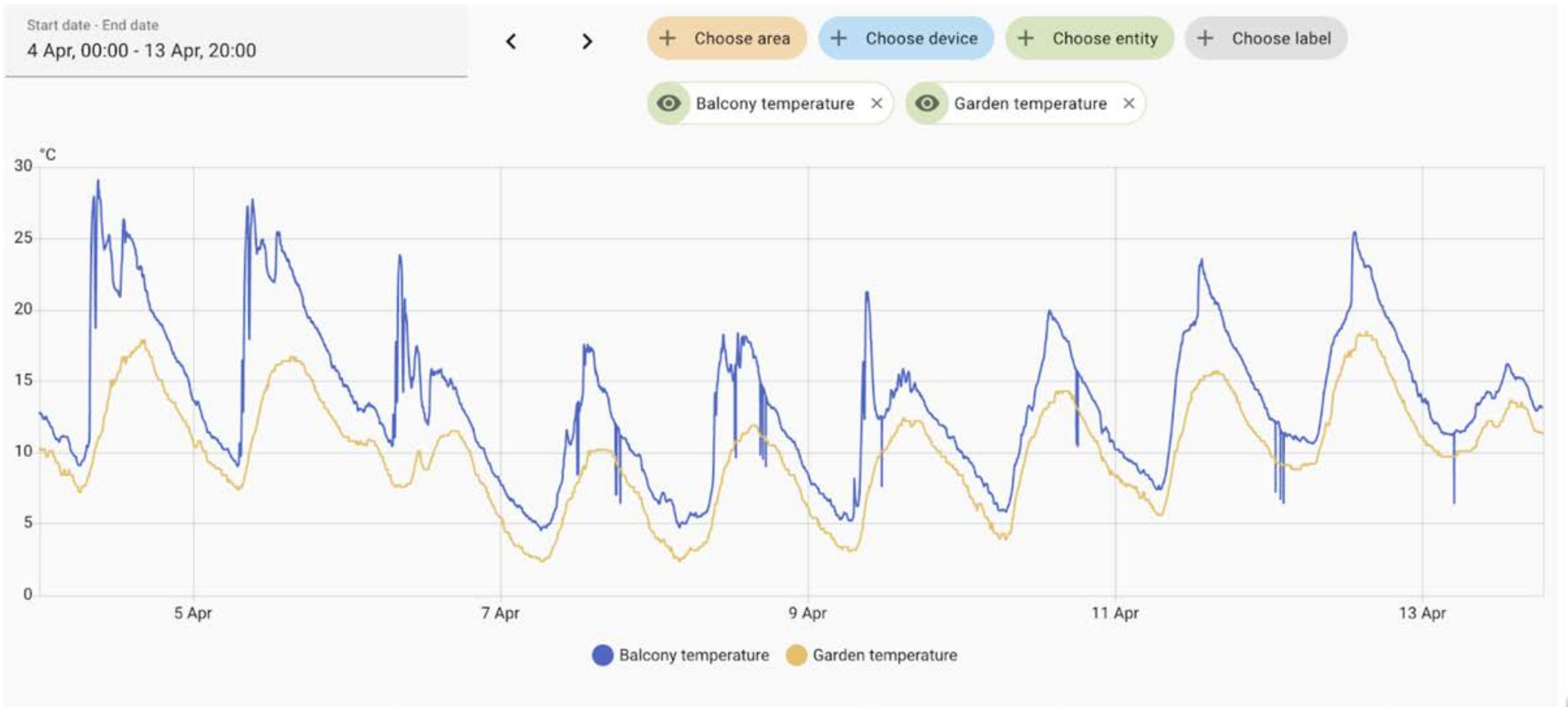
The main rack is located in the basement, which is why the radio signal there was poor. I placed the SDR on the first floor and connected it to the existing pfSense router. That

involved manually installing packages and struggling to have it start at boot. The pragmatic solution was to start a screen session with `rtl_433` running in it.

That worked; the data was sent to MQTT, and Home Assistant was plotting nice graphs of the temperature sensors I selected. This solution was not very stable. After each reboot, upgrade, or firewall failure, I had to remember to restart it. It was not reliable, and I was not satisfied with it.



Home assistant smoothly plots the temperature captured by the Garden temperature sensor.



Plot of the temperatures of both sensors captured by the Home Assistant system.

The second implementation

I had a Pine A64+ lying around with no real purpose. I started verifying the FreeBSD support for it, but I quickly dismissed it due to some troubles with the reboot. The issue persists after a software reboot, indicating a hardware problem. I recall that physically accessing the board and disconnecting the power supply was too much trouble and not very practical.

EuroBSDCon to the rescue

At EuroBSDCon 2019 in Norway, there was a tutorial by Tom Jones titled "An Introduction to Hardware Hacking with FreeBSD." Every participant received a nice little box containing

switches, LEDs, cables, and other components, and most importantly, an Arm SBC.

It was a NanoPi NEO LTS from FriendlyElec. After the conference, I used it for a bit, but now it is mainly stored in a drawer. So, I started evaluating it as the platform on which to run rtl_433 and acquire the radio data.

The installed software was outdated, so I acquired a new MicroSD card and, following the instructions published by Tom Jones on his pages, successfully moved the acquisition system to it. (<https://adventurist.me/posts/00297>)

It was all fun and worked correctly for a few days, but then it abruptly stopped; the system was not reachable over SSH, so I resorted to power cycling it. It would have been better to attach a serial console to it to find out what the problem was, but now, I didn't have the time to troubleshoot it properly.

Suspecting some incompatibility with the operating system release, I later wrote a crude shell script to automate the image creation based on Tom's instructions. After a few runs, I started parameterizing it to have a quicker way to change releases and added a few other nice-to-have features.

It was all fun and worked correctly for a few days, but then it abruptly stopped.



The final implementation — NanoPi with the USB SDR dongle sitting on the firewall.

Scripting

The script in its final form has a few more steps than the first version. First, we define the FreeBSD version we want to download and the source URL.

```
#!/bin/sh
VERSION="14.2"
RELEASE_VERSION="$VERSION-RELEASE"
URL_IMG="https://download.freebsd.org/releases/arm/armv7/ISO-IMAGES/$VERSION/FreeBSD-$RELEASE_VERSION-arm-armv7-GENERICSD.img.xz"
URL_CHECKSUM="https://download.freebsd.org/releases/arm/armv7/ISO-IMAGES/$VERSION/CHECKSUM.SHA512-FreeBSD-$RELEASE_VERSION-arm-armv7-GENERICSD"
```


Next, we define the target image name, the directory tree, the needed packages, the u-boot bootloader location, the time zone, the packages to be installed on the target image, and the initial target image size.

```
TARGET_IMG="nanopi-rtl-433-FreeBSD-$RELEASE_VERSION.img"
IMG_TARGET_DIR="img-target"
IMG_MNT_DIR="img-mnt"
IMG_RELEASE_DIR="img-release"
CUSTOM_DIR="customization"
PACKAGES="u-boot-nanopi_neo-2024.07"
BOOTLOADER="/usr/local/share/u-boot/u-boot-nanopi_neo/u-boot-sunxi-with-spl.bin"
TIME_ZONE="/usr/share/zoneinfo/CET"
TARGET_PACKAGES="rtl-433 monit"
SD_SIZE="6G"
```

For security reasons, we create a personalized user password.

```
# user password
TARGET_USER_PW=`date +%s | sha256sum | base64 | head -c 13`
```

The further steps involve preparing the directory tree and defining the checksum files and the image's location.

```
# prepare the directory tree
echo "Preparing the directory tree ..."
mkdir -p "$IMG_TARGET_DIR"
mkdir -p "$IMG_MNT_DIR"
mkdir -p "$IMG_RELEASE_DIR"
mkdir -p "$CUSTOM_DIR/usr/local/etc"
digest="$IMG_RELEASE_DIR/CHECKSUM.SHA512-FreeBSD-$RELEASE_VERSION-arm-armv7-GENERICSD"
image="$IMG_RELEASE_DIR/FreeBSD-$RELEASE_VERSION-arm-armv7-GENERICSD.img.xz"
digest_ext="$IMG_RELEASE_DIR/CHECKSUM.SHA512-FreeBSD-$RELEASE_VERSION-arm-armv7-GENERICSD.img"
image_ext="$IMG_RELEASE_DIR/FreeBSD-$RELEASE_VERSION-arm-armv7-GENERICSD.img"
```

The images are downloaded to the \$IMG_RELEASE_DIR, but only if the image checksum is different from the existing one.

```
if ! [ -f $digest_ext ]; then
    echo "Digest=Initial" > $digest_ext
fi
# get the release image checksum and check the existing image
echo "Checking compressed image checksum"
fetch -o "$IMG_RELEASE_DIR" "$URL_CHECKSUM"
if ! sha512 -q -c $(cut -f2 -d= $digest) $image; then
    echo "Download compressed image"
    fetch -o "$IMG_RELEASE_DIR" "$URL_IMG"
    # reset extracted image checksum
    echo "Digest=Refresh" > $digest_ext
    rm -f $image_ext
else
    echo "Compressed image OK"
fi
```


The image is extracted if its checksum differs from the existing one.

```
# extract the image
echo "Checking extracted image ..."
if ! sha512 -q -c $(cut -f2 -d= $digest_ext) $image_ext; then
    echo "Extracting image and creating checksum"
    rm -f $image_ext
    xz -dk $image
    sha512 $image_ext > $digest_ext
else
    echo "Extracted image OK"
fi
```

Now the packages required on the host system are installed.

```
# Install needed packages
echo "Installing packages"
pkg install -y $PACKAGES
```

A target image is created with the size defined above and connected to a memory disk.

```
# Create target image
echo "Create target image and connect to the memory disk"
truncate -s $SD_SIZE $IMG_TARGET_DIR/$TARGET_IMG
mdisk=`mdconfig -f $IMG_TARGET_DIR/$TARGET_IMG`
echo $mdisk
```

We write the image and the bootloader to the target.

```
# write the extracted image to the memory disk
echo "Writing the extracted image to the memory disk"
dd if=$image_ext of=/dev/$mdisk bs=1m
# write the bootloader to the memory disk
echo "Writing the bootloader to the memory disk"

if [ -f $BOOTLOADER ]; then
    dd if=$BOOTLOADER of=/dev/$mdisk bs=1k seek=8 conv=sync
else
    echo "The bootloader $BOOTLOADER does not exist. Aborting ..."
    exit
fi
```

Finally, the target image is mounted, and the packages are installed leveraging the `pkg-chroot` option, which allows us to install packages meant for a different hardware platform elegantly. Note that a correct `resolv.conf` should exist on the target system.

```
# mount the target image
echo "Mounted target image on $IMG_MNT_DIR"
mount /dev/"$mdisk"s2a $IMG_MNT_DIR

# install target packages
echo "Installing packages on target"
cp /etc/resolv.conf $IMG_MNT_DIR/etc/
pkg --chroot $IMG_MNT_DIR install -y $TARGET_PACKAGES
```


The services are being enabled on the target. Again, we use a handy option of the sysrc utility to access the configuration file directly on the mounted image. Not strictly needed, but we also enable the remote syslog, which should be enabled with "-a <peer>" on the destination host. This was enabled while debugging the system, which had been hanging for a day and a half, as mentioned above.

```
# enable services on target

sysrc -f $IMG_MNT_DIR/etc/rc.conf hostname="nanopi" ntpdate_enable="YES"
ntpd_enable="YES" rtl_433_enable="YES" monit_enable="YES" syslogd_enable="YES"
syslogd_flags="-s -v -v"
```

In the customization directory, we replicate the needed parts of the system tree:

```
customization/
├── home
│   ├── freebsd
│   │   └── .ssh
│   │       └── authorized_keys
├── usr
│   ├── local
│   │   └── etc
│   │       ├── monitrc
│   │       ├── rtl_433.conf
│   │       └── syslog.d
│   │           └── remote_anirul.d112.conf
└── var
    └── log
        └── rtl_433.log
```

```
# copy the customisations
echo "Copy $CUSTOM_DIR to $IMG_MNT_DIR"
cp -a $CUSTOM_DIR/* $IMG_MNT_DIR
```

The time zone is set by simply copying the definition to /etc/localtime on the target.

Fun note — copying the CET definition on a desktop machine will cause issues in the Mozilla products. For example, in Thunderbird, the time in the email list is in UTC, but when you open the email, the time is displayed correctly. In that case, it's better to use tzsetup and specify the correct city or copy over a city definition. I still must figure out how to report this bug.

```
# set the timezone
echo "Set the timezone: copy $IMG_MNT_DIR/$TIME_ZONE to $IMG_MNT_DIR/etc/localtime"
cp $IMG_MNT_DIR/$TIME_ZONE $IMG_MNT_DIR/etc/localtime
```

Next, the new password for the FreeBSD user is set up and copied to a file in the home directory, and next to the image file. The final cleanup unmounts the image, removes the memory disk, and provides the user with some final instructions.

```
# configure users
echo "Configuring users on target"
# change freebsd user password
echo "$TARGET_USER_PW" | pw -R $IMG_MNT_DIR mod user freebsd -h 0
```



```

echo "$TARGET_USER_PW" > $IMG_MNT_DIR/"home/freebsd/.pwd"

# unmount the target image
echo "Unmounted target image from $IMG_MNT_DIR"
umount $IMG_MNT_DIR

# create the password file
echo "$TARGET_USER_PW" > $IMG_TARGET_DIR/$TARGET_IMG".pwd"

# remove the memory disk
echo "Removing memory disk $mdisk"
mdconfig -d -u $mdisk

# Final instructions
echo "You can copy the image with \"dd if=$IMG_TARGET_DIR/$TARGET_IMG of=/dev/<USB
Disk> status=progress bs=1m\""
echo "The password for the user freebsd is \"$TARGET_USER_PW\""
echo "The password is also available in the file \"$IMG_TARGET_DIR/$TARGET_IMG".
pwd\""

```

In the customizations, we have the configuration files for the services. Monit monitors the rtl_433 status and restarts it if needed. Here are the lines from monitrc in which the rtl_433 service is defined.

```

# rtl_433
check process rtl_433 with pidfile /var/run/rtl_433.pid
start program = "/usr/local/etc/rc.d/rtl_433 start" with timeout 60 seconds
stop program = "/usr/local/etc/rc.d/rtl_433 stop"

```

And of course, we must configure rtl_433. The differences from the sample config are the following.

```

output mqtt://<homeassistant>,usr=<mqtt user>,pass=<mqtt user pwd> ,retain=0,devic-
es=rtl_433[/model][/id]
output json:/var/log/rtl_433.log
output log

```

The output is sent to the MQTT broker accessible to Home Assistant or, in my case, running in the same VM. The retain flag is set to false, as I don't want to clutter the MQTT topics with transient devices and am willing to get the devices back slower in Home Assistant.

The JSON output is also sent to a JSON file in /var/log

Lessons learned

That last line in the rtl_433 config file was the catch. In my prototypes, the log was written to /tmp, filling it in approximately 36 hours and causing the system to hang. So, never write log files to /tmp if you are not purging them.

The hardware should be reliable. I had a lot of trouble with the Pine A64+ and its reboot issues.

I learned a lot about the basic configuration of FreeBSD and once again appreciated the elegance of its tools.

The status

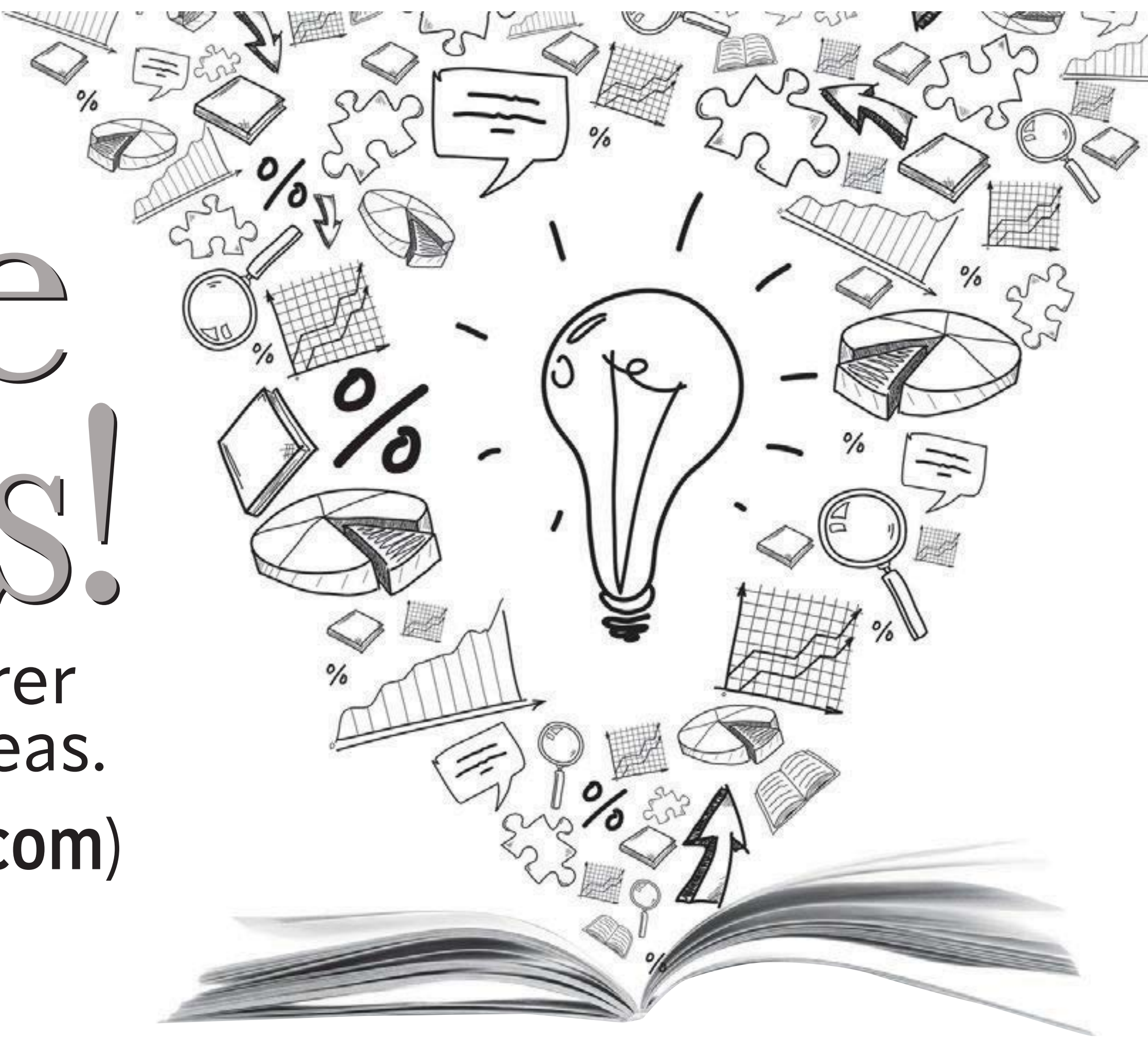
The system has maintained an uptime of 57 days since the last storm and power failure, during which it booted up automatically without user intervention. It is reliable, easily replicated, and upgradable to a new FreeBSD release. By swapping the SD card, it is trivial to test a new release or configuration and revert to the previous one.

In the future, I would like to explore the possibility of using nanobsd for development and consider whether this project could serve as a good opportunity to delve into Crochet/Poudriere image creation.

VANJA CVELBAR is a Technical Collaborator at the Istituto Officina dei Materiali of the Italian National Research Council, where he is the head of the IT group and manages the IT infrastructure and services that support research and operations. His expertise spans systems administration, networking, and open-source technologies. After two decades working primarily with Linux, he returned to his roots in FreeBSD — his first system being version 3.2. Today, he focuses on deploying and maintaining FreeBSD-based environments to deliver core IT services across the Institute.

Write
For Us!

Contact Jim Maurer
with your article ideas.
(maurer.jim@gmail.com)



Writing Effective Bug Reports

BY TOM JONES



A few years ago, our cherished former FreeBSD Journal editorial board member, Kristof Provost, wrote a great piece on the ideal way to write a bug report. Sadly, Michael W Lucas uses up our per-issue allotment of sarcasm early in the year, so to avoid going into debt, it was decided that I reframe Kristof's original post in a way that would avert ironic bankruptcy.

Every system maintainer has their own preferences for how to receive reports of bugs and requests for improvements. As Kristof is so core to pf development in FreeBSD, he was the ideal person to write a piece for the journal.

In response to being asked to reframe his original piece, Kristof responded:

"I cannot build on perfection."

So, it is up to me to rephrase Kirstof's original words and provide you with some starting advice to help you quickly sort out your favourite and most troublesome bugs. You can find his blog post at <https://www.sigsegv.be/blog/2014/M>.

Bugs

Often, software projects feel like they are made out of bugs; developers will tell you everything is made from shoe polish and duct tape. The issues that cause the most trouble are often the hardest to pin down. "Late at night, when we have 12 concurrent requests hitting our geodetic load balancer" isn't an uncommon start for a complex bug. Sometimes there is a dreaded "after 1000 hours..." lead into an issue description.

Thankfully, not all bugs are like this. FreeBSD will panic in many circumstances, and for a user, that can be a bad day, but for a developer, a panic message can be a handy clue about where things are going wrong. I'd take a panic over a silent data corruption issue any day.

Some bugs are purely cosmetic, fields aren't displayed as well as they may be, or documentation is unclear (yes, we consider that a bug!).

Whatever form your bug takes, from logical impossibility to a typo, I am going to show you a framework you can follow to get things fixed.

Whatever form your bug takes, from logical impossibility to a typo, I am going to show you a framework you can follow to get things fixed.

The bug life cycle

Before we go into the best way to write up a bug, let's discuss what happens afterward, as I think this provides the context for why you need to add as much information as you can to initial reports.

In FreeBSD, most bugs propagate either via mailing lists or the bug tracker (bugs.freebsd.org). Developers regularly read the mailing lists, but the best way to get attention for a bug is a ticket on the bug tracker.

A newly created bug is marked as "new" and is in what we call the new bug state. At this point, it has been submitted by a user, but no relevant project areas have been alerted to its existence.

The Bugmeister team goes through new submissions and reassigns them from the new bug state, making a best attempt at setting the category of the bug correctly. Developers can choose to be notified about bugs in specific categories. Relevant project mailing lists will also receive notifications about new bugs. Additionally, a summary email containing "bugs of interest to this group" is sent periodically.

Bugs may be assigned to a particular developer by Bugmeister or by other FreeBSD project members.

A developer may at some point decide that they want to "take" a bug, becoming the owner of the issue. This is typically the point at which the developer chooses to work on the bug or pursue analysis.

There may be back and forth on the issue as the developer asks for more information, but if everything goes well, eventually the developer will create a code review (on reviews.freebsd.org). The code review may be referenced in the bug, but it might be that the developer is only seeking knowledgeable feedback from certain other developers familiar with the subsystem.

If testing is required to determine if the bug is resolved, it will likely be highlighted.

Finally, once code review and testing are completed, the bug will be committed to FreeBSD; typically, the commit message will contain a reference to the bug, and an automated update will be added. Sometimes it takes many fixes to address the issue underlying a bug.

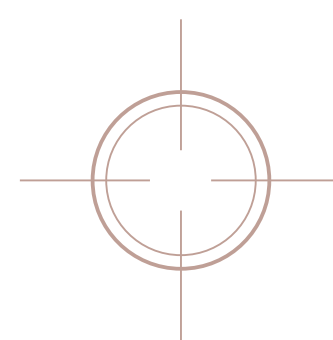
Once the fix(es) are committed, you (the user) can use the software again bug-free, assuming you are running a CURRENT snapshot. Many fixes are merged from CURRENT back to stable branches; we call this a merge from current (MFC).

If the bug is MFC'd, then the fix will be available on a stable branch and will be available in the next point release.

This process, from reported issue to fix, can involve a lot of steps; some problems can be resolved in 20 minutes, but others might take months or years. I have seen bugs closed out as fixed after more than a decade of debugging.

The time it takes to resolve an issue depends significantly on the recency of the problem. If you catch the bug soon after the commit lands, there is a good chance it will be fixed today. Other bugs take longer to show up, or only appear in environments running FreeBSD releases.

Bugs may be assigned to a particular developer by Bugmeister or by other FreeBSD project members.



The time it takes to fix those bugs largely depends on the quality of the bug report and the ongoing discussions about the issue. So let's look at what a good bug report contains.

Where to submit a bug report

The FreeBSD project encourages users to share their experiences with the software we develop through multiple channels for reporting issues. In order of preference, notifications about new or interesting bugs would come as one of the following:

- an easy-to-apply change with clear tests provided as a new code review on reviews.freebsd.org
- a hacked-up solution to an issue on reviews.freebsd.org
- a new bug on bugs.freebsd.org
- an email (or other communication) directly to the developer
- a complaint on the forums with speculation about the cause
- IRC, matrix, Discord, or another chat venue
- ~~an angry post on social media that FreeBSD is the worst~~

The first two options assume quite a high level of technical competence. If this is too much for you, don't worry; we are still happy to receive bug reports through other methods.

We don't expect everyone to do advanced software development, just that every reporter provide as much relevant detail as they can at each step.

For a typo, most users can immediately suggest a remediation, but creating a documentation change requires using developer tools, and we don't ask that of everyone. If you can use these tools, creating a review is preferred over creating a new bug, as the final change will likely require a review either way.

More complex bugs might be addressable immediately; some complex bugs have simple answers. However, other bugs are challenging to diagnose, and the best information we have about them for a long time is sideways glances as something seemingly unrelated fails.

If you have a proposed fix of any quality, the most important thing to do is to share your suggestion with the public. Work down the list until you find the correct place.

Writing a good bug report

There is a lot of information out there on what should go into a good bug report; the breadth of software in FreeBSD makes any single template pretty tricky to use.

There are some good rules of thumb for what you should include:

- Information about your setup.
- What you expect to happen.
- What actually happens.
- What is the difference between what should happen and what happens?
- This isn't always obvious.
- Events leading up to the bug being triggered.
- Temporal factors help developers debug issues. If the bug occurs immediately after a reboot or only after 5 days, different strategies are required.



The FreeBSD project encourages users to share their experiences with the software we develop through multiple channels for reporting issues.

Where to get information and what to provide

As the system runs, it generates a lot of information that can be helpful for debugging. You may be asked for logs. If you are, you should try to include things like:

- dmesg: the output of the **dmesg** command
- all of the text of a kernel panic message, including the stack trace
- the output of **pciconf -lv** if it is a hardware issue
- syslog output
- tool output and error messages

Err on the side of providing more information than less; if a developer has to ask for more information, that might add a week before they can look at the issue in detail.

Information about your setup

You need to strike a balance between describing the Anycast load balancing setup for your Fortune 500 company and simply showing a single firewall rule. Typically, reports provide less information about their environment than they could; the combination of tools and network configurations helps developers understand bugs.

So what constitutes your setup?

We need to know the version(s) of FreeBSD you are using and how many there are. The fact that the bug occurs between 14 and 15 hosts is more relevant than knowing it is just 14.

We need to know about customizations and detours from the norm; you should include them if you built FreeBSD yourself or if you are using packages. We support both, but if you aren't running the release engineering version, then that is something to consider.

If you aren't running FreeBSD, but a downstream such as pfSense or HardenedBSD, you *need* to include that. It doesn't mean FreeBSD developers won't help you, but omitting this fact will likely annoy someone.

This is relevant because downstreams build FreeBSD with different flags, packages, and ship their own tuning. All of this can contribute to a bug existing on one platform and not another.

We really need your setup and not an idealised version or an example from a tutorial.

What you expect to happen

Another essential part of reporting an issue is describing what you expect to happen. Some software can't do what you want, either now (due to feature development) or ever (due to NOT ACTUALLY BEING A BUG!).

ls won't copy files or play videos, but **pfctl** should be able to clear rule state counters. Please tell us what you expect the output and effects to be so we can double-check all of your assertions.

What actually happens?

Provide output from the tool, describe the actual effects you can measure, and highlight any issues.

What is the difference between what should have happened and what happened?

It will be evident to you that the tool is broken. You are submitting a bug, but as a developer triaging an issue, additional help in spotting what is going wrong is essential.

Many bugs aren't easy to see in a textual report, and performance issues can be challenging to relate to (and to diagnose and fix).

Be very clear about what has happened and what isn't correct.

Events leading up to the bug being triggered

For many bugs, the events leading up to the bug being triggered are important. Can't do X after Y is an excellent bug report — it implies a path to developer gold (see below).

"`ls` reports incorrect file sizes once disk is full" tells us a lot of information that "`ls` reports incorrect file sizes" doesn't.

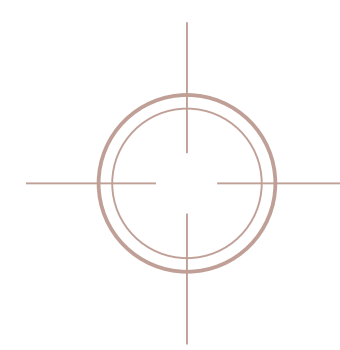
It might help to review `dmesg` before and after an instance of a bug; kernel subsystems will report errors and resource exhaustion there. In `dmesg`, we can see that the interface is going up and down constantly.

How should you provide information?

The information you include should ideally be textual. If there are error messages, you should include all of them; some software will always print error messages or print odd diagnostics when it performs operations. In a bug report, you should try to include this information.

If you are experiencing panic, you might struggle to copy and paste text from somewhere. In that case, taking a photo with your phone is a valid strategy; however, you should try to ensure a balance between file sizes and image clarity. Ideally, you would take a photo and transcribe the text as accurately as possible.

Your photo host is likely to go away in the future, but text strings in the FreeBSD bug tracker live for decades.



For many bugs, the events leading up to the bug being triggered are important.

Writing an excellent bug report

An excellent bug report will carry most of the above information, but it will also include developer gold: A reproduction case.

For a simple issue, this is something like a command invocation, but for some problems, you might have to write shell scripts that work together. Reproducers for network bugs can be very difficult to write, but thanks to `vnet` jails, it is actually possible to do this with just a shell script (that is, in fact, how the firewall test suites work).

A reproducer aims to reduce the reproduction of the problem to its barest components.

Proactively help to resolve the issue

Once you have submitted the bug, that might be enough to get a developer's interest to fix it. If you highlight a regression or new bug within 20 minutes of a commit, you are likely to get the developer's attention right away. Most developers watch mailing lists and the `src-commits` list after making changes to catch exactly these reports.

If your bug languishes or only gets assigned to a team after a while, don't panic; that is a pretty usual experience. You can proactively contact developers who work in similar areas or on similar ports. Just be polite; you are using a volunteer's time.

Once a developer has picked up your bug, they will likely ask you questions. If it has been a long time, that question might be, "Does this still happen on a snapshot?"

Otherwise, they may ask for more information or for you to test patches. If the bug is only on specific hardware, we can frequently have a fix sitting in the bug tracker, but without

someone to confirm it solves the bug, or at least doesn't cause a regression, the patch will languish.

Some bugs are just difficult to figure out. If a developer is working with you, feel free to suggest your theories for what might be the trigger. Some bugs sound like ghost stories until we find an eventual fix.

Good luck!

Writing and submitting a bug report is a lot of work; it is easy to feel frustrated that your effort seems ignored or not followed up on. FreeBSD is a volunteer project, and developers work on things as they can based on their interests. We have a great set of developers who will hunt down obscure and complex issues, but they need your help and cooperation to gather enough information to reproduce and test changes.

If you follow the advice here, you should have a much better experience getting exposure for your bugs and getting them fixed.

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



The FreeBSD Project is looking for

- Programmers • Testers
- Researchers • Tech writers
- Anyone who wants to get involved

Find out more by

Checking out our website
freebsd.org/projects/newbies.html

Downloading the Software
freebsd.org/where.html

We're a welcoming community looking for people like you to help continue developing this robust operating system. Join us!

Already involved?

Don't forget to check out the latest grant opportunities at freebsd.foundation.org

**Help Create the Future.
Join the FreeBSD Project!**

FreeBSD is internationally recognized as an innovative leader in providing a high-performance, secure, and stable operating system.

Not only is FreeBSD easy to install, but it runs a huge number of applications, offers powerful solutions, and cutting edge features. The best part? It's FREE of charge and comes with full source code.

Did you know that working with a mature, open source project is an excellent way to gain new skills, network with other professionals, and differentiate yourself in a competitive job market? Don't miss this opportunity to work with a diverse and committed community bringing about a better world powered by FreeBSD.

The FreeBSD Community is proudly supported by



Implementing a Quantum-Safe Website on FreeBSD

BY GERGELY POÓR

Some time ago, at my current workplace, it was brought to my attention that traditional cryptography will no longer be considered secure.

According to various sources, it is estimated that in the next 10 years, the advancements in quantum computing will reach a point where modern-day cryptographic algorithms that we take for granted will be easily broken within seconds. Naturally, I started researching this topic to find out how bad the situation is and what we can do today to prepare our systems for this scenario. This so-called "Quantum Threat" is generally described as the point in the future where quantum computers will possess enough processing power to break traditional encryption in a matter of minutes or even seconds. Sounds bad, right? Well, it gets even worse. There is a related phenomenon called "Store now - decrypt later" or "Harvest now - decrypt later," which basically means that currently secure data transmissions sent over the internet are captured and stored until a sufficiently powerful quantum computer is widely available for anyone to utilize it to decrypt the previously captured data.

But what is "currently secure"? In terms of asymmetric encryption, there are key exchanges, the most used method of which is RSA using 2048, 3072, or 4096 bits of key length. These key exchanges (not limited to only RSA) can be a part of an SSH session to a remote server, using a TLS-encrypted website to enter your bank account details, or even an IKEv2-based IPSec VPN tunnel between two remote locations. All the data sent over these channels can be intercepted and stored now and decrypted later. But how does one go about decrypting RSA keys? The answer to that question is Shor's algorithm, which was invented by Peter Shor in the 1990s. It was designed to find the factors of a prime number using a quantum computer. But what does this have to do with key exchange algorithms? Let's take RSA, for example. In simple terms, it works by multiplying two big prime numbers to create an even bigger number. The two primes are kept secret and are part of the private key alongside some other numbers. The product of those two primes is part of the public key with some extra numbers. What makes this vulnerable is the fact that if you manage to find out the two starting prime numbers, you can use them to compute part of the private key, which can be used to find the remainder of it. If you have the private key, you can decrypt what one side sent. Do it again for the other, and you have yourself a fully decrypted captured information exchange. Of course, it takes a tremendous amount of computing power

It was brought to my attention that traditional cryptography will no longer be considered secure.

because the bigger the public key is, the more number-crunching you must do. Shor's algorithm provides a method to speed up the process using a quantum computer, which can calculate all the possible solutions at once.

But how can we defend our systems against this kind of attack? There are three answers:

You can keep using the standard algorithms, but with longer keys. Currently, it is believed that RSA keys over 4096 bits of key length are considered long enough that they cannot be cracked soon. If you want to play it safe, use key lengths of 8192 bits or longer.

However, there are other alternative methods to tackle this problem. The second one is PQC (Post-Quantum Cryptography). Researchers and mathematicians started working on other algorithms that would be resistant to Shor's. These were based on so-called mathematical "trapdoor" functions: easy to get from an equation to a result, but (almost) impossible to do in reverse. The United States' National Institute of Standards and Technology (NIST) began collecting these algorithms, and numerous cryptography experts and mathematicians started testing them to find out whether they are resistant to Shor's algorithm or not. This was known as the Post-Quantum Cryptography project. Over the years, there were several rounds where various algorithms were eliminated. In 2024, NIST issued the FIPS 203 standard naming ML-KEM (previously known as CRYSTALS-Kyber or Kyber) as the primary standard for general encryption, and in 2025 stated that HQC will be a backup in case ML-KEM is ever compromised.

But how can we defend
our systems against
this kind of attack?
There are three answers:

The third option is called QKD, which stands for Quantum Key Distribution and is based on the manipulation of photon particles using quantum physics to securely generate keys. This method is extremely costly as it requires specialized equipment and a pre-existing optical network connection between the two participating sides and has a current limit of around 100km between the endpoints.

The project's goal

I have a website that I made some time ago using only nginx without any kind of HTML, CSS, PHP, or whatever. It's a simple website that returns the IP address of the client (like icanhazip.com or ifconfig.me). It started out as a hobby project while I was learning about nginx, but I started to deploy it in larger networks to test if the client was behind NAT or not. This was a good starting point for my PQC experiment. The requirements were simple: it must work with a wide range of software (web browsers, command-line tools like `fetch(1)` or `curl(1)`), and having experienced the UNIX philosophy during my years of working with FreeBSD and Linux I wanted to keep it as simple as possible but stable as well since it could also run on a cloud VPS at some point in the future. So naturally, I chose FreeBSD as the OS and nginx as the platform. The only remaining part was the actual PQC implementation.

I did a little research and found a project called [oqs-provider](#) by the Open Quantum Safe project, which is an open-source C library and provider for OpenSSL version 3, implementing ML-KEM among other algorithms. It is available for FreeBSD and for various Linux distributions as well.

How it works

Simply put, it integrates various PQC algorithms for key exchange and signature with OpenSSL. In terms of key exchanges (alongside others), it supports ML-KEM with several elliptic curve-based Diffie-Hellman key exchanges like X25519, p384, p521, and SecP384r1. These can be easily identified based on their names, like X25519MLKEM768, which uses Curve 25519 with 768-bit long ML-KEM keys. PQC algorithms require TLSv1.3, but for compatibility reasons, we will define TLSv1.2 as the minimum version, so legacy systems will still be able to reach the website. One thing to keep in mind is the [“TL;DR fail”](#) error, which can happen if the client software is not properly set up to support PQC algorithms over TLSv1.3, resulting in a TLS failure, but this will gradually be a smaller nuisance over time as software gets updated and rolled out to clients. If you are not concerned with legacy clients or buggy software and want a 100% quantum-safe website, feel free to disable TLSv1.2 altogether (as you’ll see in the nginx configuration file later). With the theory part out of the way, let’s get to the fun stuff: the actual implementation!

Implementation

The oqsprovider requires OpenSSL version 3.2 or higher. According to their GitHub page, they have added some extra functionality starting with version 3.4. My FreeBSD installation has OpenSSL version 3.0.16, which doesn’t support oqsprovider.

At the time of this writing, OpenSSL 3.5.0 was released with native PQC algorithm support, but a note from pkg(8) indicated that it was in beta stage, not suitable for production. So, for the rest of the implementation, I will stick with OpenSSL version 3.4.1.

To start things off, I updated my VM to FreeBSD 14.3-RELEASE. We will have to install a newer version of OpenSSL as well as nginx, but with the ability to use the newer OpenSSL. To do this as hassle-free as possible, we will install openssl34 and openssl-oqsprovider via pkg(8), and nginx will be built using the ports system. For this reason, we will need to have the ports tree present under /usr/ports. I don’t have security/openssl34 present on my system, so I will be pulling the 2025Q2 branch of the ports tree. I will need that so nginx can be linked against openssl34. First, I will install git(1), which is the recommended method to install/update the ports tree as stated by the FreeBSD handbook.

Simply put, it integrates various PQC algorithms for key exchange and signature with OpenSSL.

```
# pkg install -y git
```

Once git(1) is on the system, it can manage the ports tree. However, I installed the ports tree with the base system some time ago, so I will be removing the current /usr/ports directory, so when I clone the repository, it will not complain about /usr/ports being already present. There are other ways around this, but I like to start things off clean.

```
# rm -rf /usr/ports
# git clone --depth 1 https://git.FreeBSD.org/ports.git -b 2025Q2 /usr/ports
```

Afterwards, we need to install OpenSSL 3.4.1 and oqsprovider.

```
# pkg install -y openssl34 openssl-oqsprovider
```

Then, as the installation message suggests, we will need to merge the contents of `/usr/local/openssl/oqsprovider.cnf` with `/usr/local/openssl/openssl.cnf`. Since we just installed the new version of OpenSSL, after the merge, the contents of `/usr/local/openssl/openssl.cnf` will look like this:

```
...
[provider_sect]
default = default_sect
oqsprovider = oqsprovider_sect
...
[default_sect]
activate = 1

[oqsprovider_sect]
activate = 1
module = /usr/local/lib/openssl-modules/oqsprovider.so
...
```

Now we will compile nginx.

```
# cd /usr/ports/www/nginx
```

I will export some environmental variables to make nginx link against the newly installed OpenSSL 3.4.1

```
# export OPENSSL_BASE=/usr/local
# export OPENSSL_LIBS="-L/usr/local/lib"
# export OPENSSL_CFLAGS="-I/usr/local/include"
```

Then we will configure nginx to make sure that "HTTP_SSL" is supported (it should be enabled by default, but it's always better to double-check). I will not adjust any other settings.

```
# make config
```

Now we are ready to start compiling nginx. Set some environmental variables to indicate the path of the newly installed OpenSSL and hit enter.

```
# make OPENSSLBASE=/usr/local OPENSSLDIR=/usr/local/openssl install clean
```

While nginx is compiling, go and grab your favorite beverage, work on some tickets, or show the compilation output to your friends so they can see how cool you are.

After the compilation is done, let's verify that nginx now links against OpenSSL 3.4.1 that we installed under `/usr/local`:

```
# nginx -V 2>&1 | grep -i openssl
built with OpenSSL 3.4.1 11 Feb 2025
# ldd /usr/local/sbin/nginx | grep ssl
libssl.so.16 => /usr/local/lib/libssl.so.16 (0x16a83b849000)
```

Note: the hex identifier in parentheses may differ.

If your output is the same as mine, you have successfully added OpenSSL 3.4 support for nginx. Next, we will create a configuration file for our website to include PQC. Let's head to /usr/local/etc/nginx, where we will first make a backup of the original nginx.conf file:

```
# cd /usr/local/etc/nginx
# mv nginx.conf nginx.conf.orig
```

And now let's create a new configuration:

```
# vi nginx.conf
```

Add the following lines to the file:

```
events{}
http{
    server{
        listen 443 ssl;
        ssl_certificate /usr/local/etc/nginx/server.crt;
        ssl_certificate_key /usr/local/etc/nginx/private.key;
        ssl_protocols TLSv1.2 TLSv1.3; #remove TLSv1.2 if you don't need backwards
compatibility
        ssl_prefer_server_ciphers off;
        ssl_ciphers ECDHE-ECDSA-AES128-GCM-SHA256:ECDHE-RSA-AES128-GCM-SHA256:ECD-
HE-ECDSA-AES256-GCM-SHA384:ECDHE-RSA-AES256-GCM-SHA384:ECDHE-ECDSA-CHACHA20-POLY1305-
:ECDHE-RSA-CHACHA20-POLY1305:DHE-RSA-AES128-GCM-SHA256:DHE-RSA-AES256-GCM-SHA384:D-
HE-RSA-CHACHA20-POLY1305; #if using pure TLSv1.3 you can remove this line according to
Mozilla's SSL Configuration Generator's "modern" settings
        ssl_ecdh_curve X25519MLKEM768:X25519:prime256v1:secp384r1; #this is the PQC
part, the others are for non-pqc compatibility only.
        location /{
            return 200 "$remote_addr\n";
        }
    }
    server{
        listen 80;
        location /{
            return 301 https://$host$request_uri;
        }
    }
}
```

This configuration will do the following:

- listen on ports 80 and 443
- redirect plaintext HTTP requests to HTTPS
- utilize TLS1.3 and TLS1.2 for compatibility
- use balanced ciphers for the widest compatibility while still providing decent security. I got the cipher and curve list from the Mozilla SSL Configuration Generator.

Next, for this demo, I will create a self-signed certificate, but for production use (and for compatibility reasons), you should acquire a valid certificate that is signed by a trusted CA. You can use Let's Encrypt certificates and automate the certificate renewal with certbot(1).

For that, you can simply run:

```
# pkg install -y py311-certbot
```

After that, follow the walkthrough found at Certbot's instructions.

To create the self-signed certificates, you can use this one-liner (remember to change the DNS and IP fields to match your setup):

```
# openssl req -x509 -nodes -days 365 -newkey rsa:2048 \
-keyout private.key -out server.crt \
-subj "/CN=quantum" \
-addext "subjectAltName=DNS:quantum,IP:192.168.2.40"
```

It will use a 2048-bit long RSA key, but you might want to bump this all the way to 8192 (remember keys over 4K are considered quantum-safe for now), but keep in mind that could break compatibility with some older systems, and will be valid for 1 year, which is more than enough for this testing purpose. (Here, the phrase "Nothing is more permanent than a temporary workaround" comes to my mind). It also includes the IP address and the hostname of my FreeBSD VM. If this were not the case, some software (like PowerShell) would complain that it cannot trust the certificate and would not be able to initiate/complete the TLS session with the website.

Once the certificate and its private key are both ready, you can start nginx, but first, we need to tell /etc/rc.conf that we would like to start it at boot time.

```
# sysrc nginx_enable=YES
# service nginx start
```

It will verify the syntax of the nginx.conf and test our configuration briefly before starting it. If everything is fine, you should see this:

```
Performing sanity check on nginx configuration:
nginx: the configuration file /usr/local/etc/nginx/nginx.conf syntax is ok
nginx: configuration file /usr/local/etc/nginx/nginx.conf test is successful
Starting nginx.
```

Testing

Now that we have a website up and running, let's check if it works. For the tests, I will use several methods: a web browser and various command-line clients, like curl(1). I have curl installed already, but if that is not the case for you, install it with pkg:

```
# pkg install -y curl
```

You can check the website with curl (keep in mind the self-signed certificate — hence we'll use the --insecure flag).

```
# curl --insecure https://127.0.0.1
```

It will return 127.0.0.1 and a newline character. To get a bit more info, you can throw in a -v flag to make the output verbose.

```
# curl --insecure -v https://127.0.0.1
```

In my case, since curl links against the default OpenSSL version, it doesn't support PQC algorithms, so it falls back to X25519:

```
...
* SSL connection using TLSv1.3 / TLS_AES_256_GCM_SHA384 / X25519 / RSASSA-PSS
...
```

You can also verify the redirect with this command:

```
# curl --insecure -vL http://127.0.0.1
```

If you see this, it worked:

```
...
* Request completely sent off
< HTTP/1.1 301 Moved Permanently
...
* Clear auth, redirects to port from 80 to 443
...
```

For Microsoft Windows-based hosts, if you are using a self-signed certificate, you'll need to import that (in our case, it's server.crt) to the "Trusted Root Certification Authorities" store and run the following PowerShell command:

```
(Invoke-WebRequest https://192.168.2.40).Content
```

Or if you want a bit more verbosity:

```
Invoke-WebRequest https://192.168.2.40
```

Using curl also works, but it's just a frontend to the Invoke-WebRequest and doesn't have the same flags as the FreeBSD or Linux versions.

To test compatibility with other hardware, I have logged on to my MikroTik router and called the URL from the command line:

```
/tool fetch url="https://192.168.2.40" output=user check-certificate=no
status: finished
downloaded: 0KiB
total: 0KiB
duration: 1s
data: 192.168.2.1n
```

Note the "n" at the end. If you want to get rid of it, change the following line in /usr/local/etc/nginx/nginx.conf:

```
return 200 "$remote_addr\n";
```

to this:

```
return 200 "$remote_addr";
```

This will not return a newline character if called from a tool like curl. Decide which version you want. If you plan to use this IP address in scripts, get rid of the "\n". For me, it is currently just for debugging, so I'll leave it as-is.

In the case of web browsers, you might need to enable some features in certain versions

if you wish to have PQC support. I am testing on Firefox version 139.0.4, which has PQC support enabled since version 132. In the case of Chrome, it has PQC support since version 124, but in some cases, you'll need to enable it by hand:

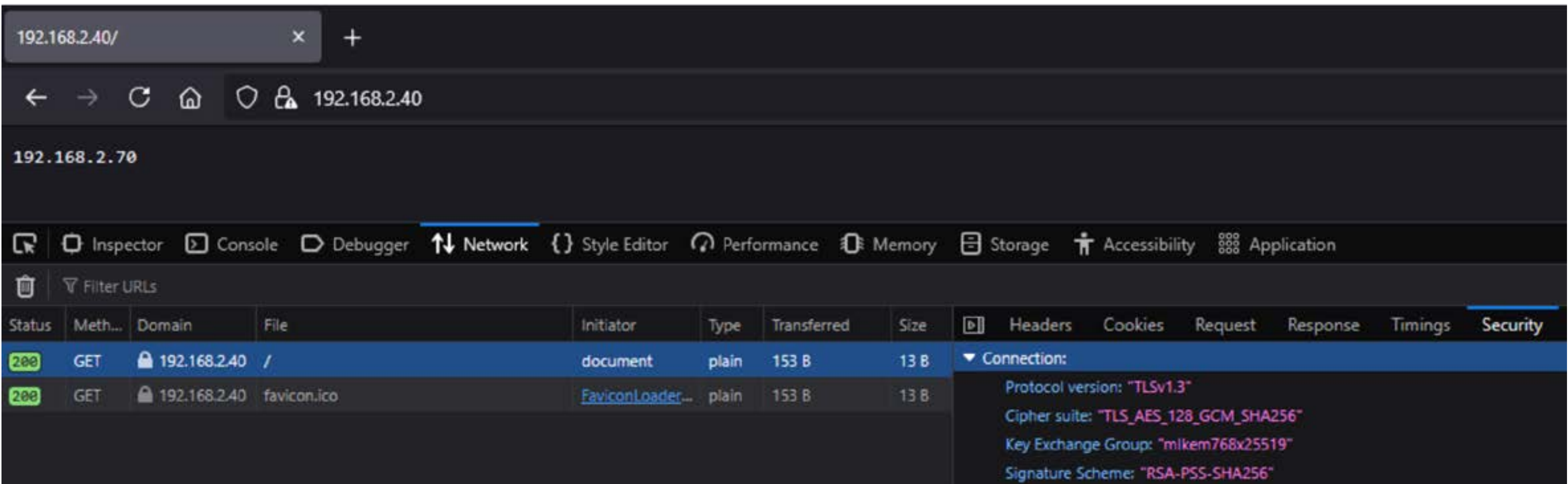
```
chrome://flags/#enable-tls13-kyber
```

You can check if Firefox supports it by going to about:config and looking for this:

```
security.tls.enable_kyber
```

Next, open the developer tools by pressing F12 and navigate to "network" if using Firefox or "Privacy and security" in case of Chrome. Then input the IP address of your FreeBSD installation and press enter. You should see only an IP address on screen (that's where the request originated from). In the developer tools, click on the entry that has your FreeBSD's IP address, and if using Firefox, also click on the "Security" tab on the right side.

If your browser supports PQC, you should see that the key exchange uses either mlkem768x25519 on Firefox or X25519MLKEM768 on Chrome.



If that's the case, then congratulations! You have successfully deployed a PQC-secured website with legacy support on FreeBSD. Welcome to the future!

Conclusion

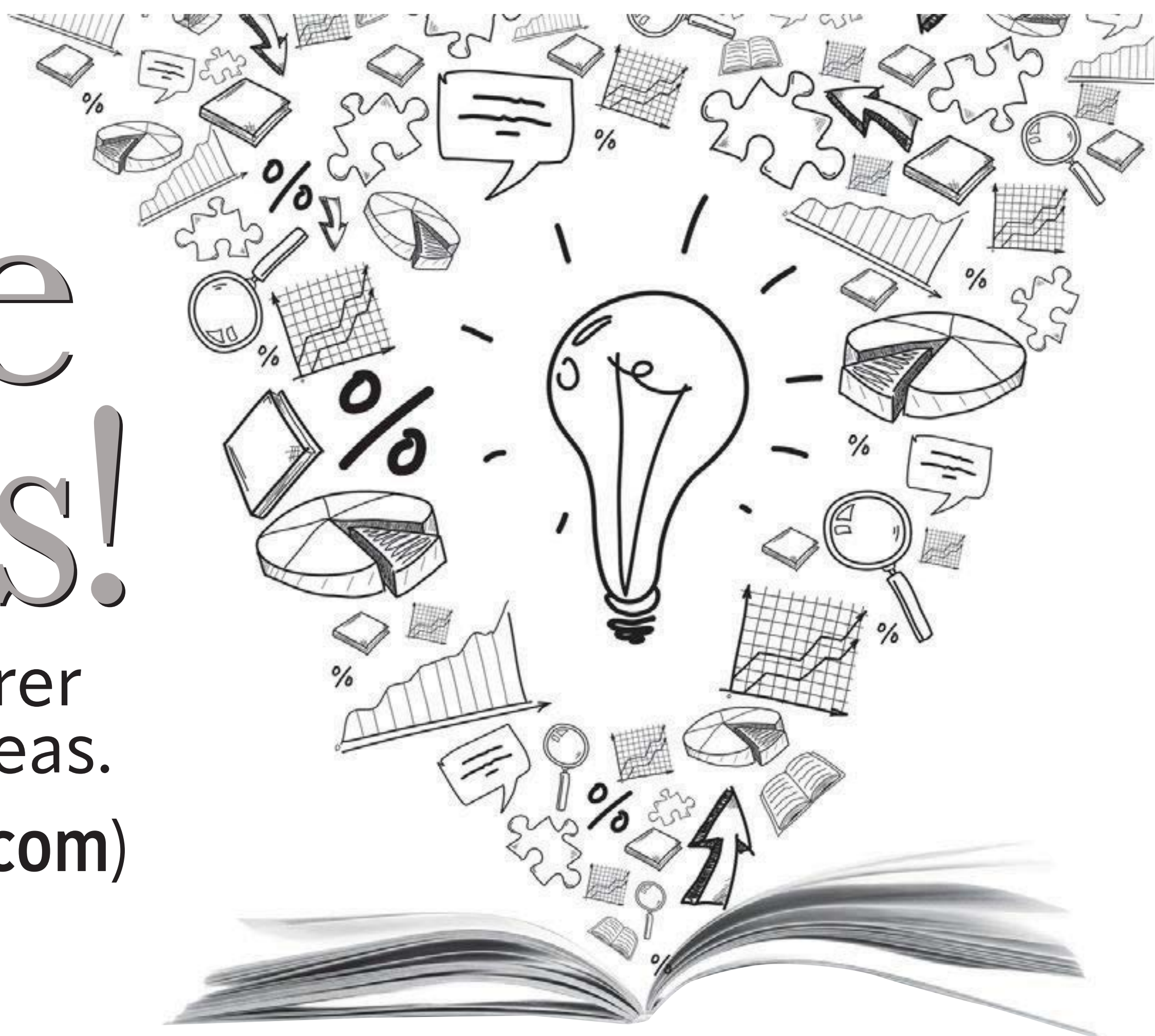
While it is not so difficult to add PQC support for a TLS key exchange, the overall process does include several extra — but needed — steps. One key concern is that you must compile nginx from source to use the newer OpenSSL version. This means that every time a security patch comes out, you will have to recompile it, which takes more time than simply applying the hotfix with either freebsd-update(8) or pkg(8). However, OpenSSL 3.5.0, which adds native support for various PQC algorithms, was recently released, and being a long-term stable (LTS) release, it will be supported until 2030 according to their website. With the ever-present quantum threat and the sudden rush to implement PQC as soon as possible, I would welcome it if this version were integrated into the FreeBSD base. That would eliminate most of the steps to get PQC working with nginx (and possibly other software). I also know, however, that stability is a major concern for FreeBSD, and until OpenSSL 3.5 has been thoroughly tested and vetted for bugs, we will probably find an older version of OpenSSL in the base install. This website was only a small example of quantum-safe encryption, but it is not hard to imagine additional software benefiting from PQC. Let's say banking, healthcare, or governmental sites started rolling out ML-KEM. It would be almost impossible to decrypt communications between the servers and the clients, so bank account

details, patient information, and personally identifiable information would be safe in transit against future quantum computers. It is not going to happen anytime soon, but as more people encounter the term “quantum-threat”, the more awareness is raised and the closer we all get to a world where post-quantum cryptography is part of our everyday lives.

GERGELY POÓR is a Linux/BSD System and Network Engineer, an Electrician, and a FreeBSD enthusiast who has been working in IT since he graduated from high school in 2018. Having experience ranging from SMB desktop support to enterprise-level hybrid-cloud and industrial/IoT systems, he is always keen on learning something new. He lives in Budapest, Hungary, with his beloved wife Kriszti and likes to program in sh/bash and develop his own smart home system in his free time.

Write For Us!

Contact Jim Maurer
with your article ideas.
(maurer.jim@gmail.com)





FreeBSD WiFi Development

Part 2: Working on a Driver

BY TOM JONES

This is the second article in a series on hacking on WiFi in FreeBSD. In the [first article](#), we introduced some terminology around WiFi/80211 networks, gave a crash course in a typical network architecture, and presented examples of using `ifconfig` and some wireless adapters to create station, host ap, and monitor mode WLAN interfaces. We also introduced the two distinct kernel layers that implement the WiFi subsystem — drivers and net80211.

Drivers Things like `iwx`, `rtwn`, and `ath`, which speak to wireless adapters over a physical bus such as USB or PCIe, usually via a firmware interface.

net80211 The abstract state machines required to join networks, send packets, and perform other complex operations are common across many drivers.

To enable flexibility in what hardware is required to provide, the net80211 layer can implement most parts of the IEEE 802.11 state machine itself. This architecture allows us to have a standard interface to integrate with the rest of the networking stack, abstracting away the specifics of what a card can support. It also enables purely software-based WLAN adapters, which can be very useful for creating test environments.

Network Adaptors can provide various levels of support from a FullMAC interface, where almost all of the processing is handled on the card directly to devices where all most all support is handled by the net80211 stack and the card just manages the radios. In a FullMAC card, the firmware exposes a configuration interface for the OS driver to use. All packet transmission, reception, and management operations, such as moving channel or scanning, are handled by the card's firmware. The `bwfm` Broadcom driver in OpenBSD and NetBSD is an example of a FullMAC driver.

Other cards need the net80211 stack to provide a variety of services to support the driver's operations. Some devices, like `iwx`, provide an interface to management operations such as scanning and joining networks, but most other operations are handled by net80211.

Older drivers have to implement more of the net80211 state machine themselves rather than reproducing a lot of very similar code.

Network Adaptors can provide various levels of support from a FullMAC interface.

All WiFi drivers exist at some point on this scale, ranging from firmware handling almost all the work to the OS managing most of the radio and transmission. The best way to see how this works practically is to look at a driver.

Let's start by looking at how a driver attaches and appears in the `net.wlan.devices` list, and then we will look into how a packet leaves the net80211 stack and heads toward the WiFi radio.

In this article, we will primarily focus on the `if_iwx` driver for a couple of reasons: I am very familiar with it, having brought the driver into the tree from the Future Crew source release, and as a new driver, there is still a lot to do in terms of low-hanging fruit.

Connecting a driver to hardware

The life cycle of a driver is usually:

- probe
- attach <do some work>
- detach

Many drivers only ever detach when the machine is turned off. The probe and attach phases are where we need to start to add support for new hardware to an existing driver or when adding a new driver.

When a device is discovered by a bus, the bus will ask each registered driver if it can work with that hardware. Once all drivers have been queried, the bus will, in probe response order, ask the driver if it can **attach** to the device. The first device to attach **wins**.

At some point in the future, the driver will have to stop playing and go home. This can happen due to a bus error, if a device is removed, such as a USB, or if the system shuts down or restarts.

For each of these phases, a callback is registered with the bus. As an example here is the `pci_methods` struct from `if_iwx.c`

```
static device_method_t iwx_pci_methods[] = {
    /* Device interface */
    DEVMETHOD(device_probe,      iwx_probe),
    DEVMETHOD(device_attach,     iwx_attach),
    DEVMETHOD(device_detach,     iwx_detach),
    DEVMETHOD(device_suspend,    iwx_suspend),
    DEVMETHOD(device_resume,     iwx_resume),

    DEVMETHOD_END

};
```

`if_iwx` registers probe, attach, and detach methods, and suspend and resume methods. All of which are called when needed.

Probe

WiFi devices are typically made from a chipset and some supporting hardware. The chipset is made by a company such as Realtek or Intel, but the actual device is manufac-

.....

All WiFi drivers exist at some point on this scale, ranging from firmware handling almost all the work to the OS managing most of the radio and transmission.

tured around the chipset by another company. This arrangement means we get rtwn-based devices made by a company such as TP-Link. The company building a device around the chipset provides drivers and configuration data, resulting in a larger number of device IDs being supported by a smaller number of drivers.

This means that a very common first patch for a new FreeBSD contributor is to add a device ID for something not yet covered (my first patch was a flash chip ID in a MIPS router!).

Your first change in FreeBSD WiFi could be straightforward, buy a device you think should work and test it (follow the instructions from the first article in this series).

If no driver probes for the hardware, you can list out the USB or PCIe device IDs and use those to determine from other platforms which driver should support them.

Two recent changes I committed to FreeBSD for an external contributor were just this, adding device IDs for hardware supported by `if_run` and `if_rum`. The driver part for the run change was:

```
diff --git a/sys/dev/usb/wlan/if_run.c b/sys/dev/usb/wlan/if_run.c
index 00e005fd7d4d..97c790dd5b81 100644
--- a/sys/dev/usb/wlan/if_run.c
+++ b/sys/dev/usb/wlan/if_run.c
@@ -324,6 +324,7 @@ static const STRUCT_USB_HOST_ID run_devs[] = {
     RUN_DEV(SITECOMEU, RT2870_3),
     RUN_DEV(SITECOMEU, RT2870_4),
     RUN_DEV(SITECOMEU, RT3070),
+    RUN_DEV(SITECOMEU, RT3070_1),
     RUN_DEV(SITECOMEU, RT3070_2),
     RUN_DEV(SITECOMEU, RT3070_3),
     RUN_DEV(SITECOMEU, RT3070_4),
```

Your first FreeBSD change could be as simple as adding a single line to the device IDs for a device. Once you have that, you need to add an entry to the relevant driver, test it, email me thj@freebsd.org a diff, and I will commit it.

Attaching to net80211

The state required for a WiFi driver is stored in an `ieee80211com` variable (usually called the `ic`) on the driver's softc.

A driver uses the IC to set flags for capabilities and override function pointers to hook or replace default functionality provided by the net80211 stack.

In the previous article in this series, I showed you how to create virtual WLAN interfaces (VAPs) on top of a driver using the `ifconfig` command. VAPs allow us to have multiple interfaces on top of a single card operating in different modes, sta, host ap, monitor, etc. The driver manages the availability of each of these modes using the `ic_caps` bit field.

The values in this field are set as part of the driver attach process. Here is an example from the `iw_x_attach` function from the `if_iwx` driver:

```
...
ic->ic_softc = sc;
ic->ic_name = device_get_nameunit(sc->sc_dev);
ic->ic_phytype = IEEE80211_T_OFDM; /* not only, but not used */
ic->ic_opmode = IEEE80211_M_STA; /* default to BSS mode */
```



```

/* Set device capabilities. */
ic->ic_caps =
    IEEE80211_C_STA |
    IEEE80211_C_MONITOR |
    IEEE80211_C_WPA |           /* WPA/RSN */
    IEEE80211_C_WME |
    IEEE80211_C_PMGT |
    IEEE80211_C_SHSLOT |       /* short slot time supported */
    IEEE80211_C_SHPREAMBLE |   /* short preamble supported */
    IEEE80211_C_BGSCAN         /* capable of bg scanning */
    ...

```

[attach from if_iwx](#)

This snippet of code resides near the end of the attach method in `if_iwx`. The preceding attach code performs driver housekeeping state with independent tasks, discovering which PCIe device this is, and determining the exact Intel Wireless model of the card.

`if_iwx` supports the station mode (`IEEE80211_C_STA`) and monitor modes (`IEEE80211_C_MONITOR`); if the driver supported hosts AP mode (as `rtwn` does), it would have the additional `IEEE80211_C_HOSTAP` flag in its capability bit mask.

Beyond modes `iwx` supports: WPA encryption (`IEEE80211_C_WPA`), multimedia extensions for differential service (`IEEE80211_C_WME`), power management (`IEEE80211_C_PMGT`), short time slots (`IEEE80211_C_SHSLOT`), preambles (`IEEE80211_C_SHPREAMBLE`), and background scanning (`IEEE80211_C_BGSCAN`).

The complete list of capabilities lives in the `ieee80211.h` header files. The capabilities a driver can advertise depend on both hardware features and support in the driver. While a driver is in development, it might not yet implement features such as WPA offload, so just because a flag is missing in a driver, it doesn't mean the hardware feature is unavailable.

The second task performed by the driver attachment phase is to take over or implement `net80211` functions, which is done through the `iwx_attach_hook` configuration callback. Here, the driver overrides function pointers for a lot of the features advertised by the ``ic_caps`` bit field.

First, `if_iwx` creates the channel map. For this card, the driver must ask the card's firmware to provide a set of supported channels.

```

iwx_init_channel_map(ic, IEEE80211_CHAN_MAX, &ic->ic_nchans,
    ic->ic_channels);

ieee80211_ifattach(ic);
ic->ic_vap_create = iwx_vap_create;
ic->ic_vap_delete = iwx_vap_delete;
ic->ic_raw_xmit = iwx_raw_xmit;
ic->ic_node_alloc = iwx_node_alloc;
ic->ic_scan_start = iwx_scan_start;
ic->ic_scan_end = iwx_scan_end;
ic->ic_update_mcast = iwx_update_mcast;

```



```

ic->ic_getradiocaps = iwx_init_channel_map;

ic->ic_set_channel = iwx_set_channel;
ic->ic_scan_curchan = iwx_scan_curchan;
ic->ic_scan_mindwell = iwx_scan_mindwell;
ic->ic_wme.wme_update = iwx_wme_update;
ic->ic_parent = iwx_parent;
ic->ic_transmit = iwx_transmit;

sc->sc_ampdu_rx_start = ic->ic_ampdu_rx_start;
ic->ic_ampdu_rx_start = iwx_ampdu_rx_start;
sc->sc_ampdu_rx_stop = ic->ic_ampdu_rx_stop;
ic->ic_ampdu_rx_stop = iwx_ampdu_rx_stop;

sc->sc_addba_request = ic->ic_addba_request;
ic->ic_addba_request = iwx_addba_request;
sc->sc_addba_response = ic->ic_addba_response;
ic->ic_addba_response = iwx_addba_response;

iw_x_radiotap_attach(sc);
ieee80211_announce(ic);

```

Then the driver either replaces or intercepts calls that net80211 will make using the device's IC. Implementations are provided for **ic_vap_create** and **ic_raw_xmit**, but other calls, such as **sc_ampdu_rx_start** and **stop**, are intercepted.

Finally, the driver attaches to the radiotap subsystems, which allows raw packets to be fed to BPF and then announces the existence of the driver to the net80211 system.

The two **ieee80211_** calls in the attach methods are examples of our interface to the net80211 system. The first call attaches our driver to the net80211 subsystem (it is here that we get added to the list behind the **net.wlan.devices** sysctl). This makes the driver available for **ifconfig** to use.

The second call (**ieee80211_announce**) handles declaring that the device has been created; this is where we print the channel and feature support for the card.

Once the driver has attached to the net80211 subsystem, it will idle until external events cause it to move into an operating state. The next part of operating is handled by net80211, and it calls out to the hooked methods we overrode in the **attach_hook** callback.

Implementing station mode

In the first article, we created a station mode VAP for our first example. The command we ran was:

```
ifconfig wlan create wlandev iwx0
```

The **wlan** argument lets the system allocate a device number for us, and the **iwx0** tells the net80211 subsystem to use the device called **iwx0** to create this VAP.

This command is translated by **ifconfig** via a library to a **net80211_ioctl** call. The final result is net80211 calling the **ic->ic_vap_create** callback on our drivers **ic**. From above, you know that this is mapped to **iwx_vap_create**.


```

struct ieee80211vap *
iwx_vap_create(struct ieee80211com *ic, const char name[IFNAMSIZ], int unit,
    enum ieee80211_opmode opmode, int flags,
    const uint8_t bssid[IEEE80211_ADDR_LEN],
    const uint8_t mac[IEEE80211_ADDR_LEN])
{
    struct iwx_vap *ivp;
    struct ieee80211vap *vap;
    if (!TAILQ_EMPTY(&ic->ic_vaps))          /* only one at a time */
        return NULL;
    ivp = malloc(sizeof(struct iwx_vap), M_80211_VAP, M_WAITOK | M_ZERO);
    vap = &ivp->iv_vap;
    ieee80211_vap_setup(ic, vap, name, unit, opmode, flags, bssid);
    vap->iv_bmissthreshold = 10;              /* override default */
    /* Override with driver methods. */
    ivp->iv_newstate = vap->iv_newstate;
    vap->iv_newstate = iwx_newstate;

    ivp->id = IWX_DEFAULT_MACID;
    ivp->color = IWX_DEFAULT_COLOR;

    ivp->have_wme = TRUE;
    ivp->ps_disabled = FALSE;

    vap->iv_ampdu_rxmax = IEEE80211_HTCAP_MAXRXAMPDU_64K;
    vap->iv_ampdu_density = IEEE80211_HTCAP_MPDUDENSITY_4;

    /* h/w crypto support */
    vap->iv_key_alloc = iwx_key_alloc;
    vap->iv_key_delete = iwx_key_delete;
    vap->iv_key_set = iwx_key_set;
    vap->iv_key_update_begin = iwx_key_update_begin;
    vap->iv_key_update_end = iwx_key_update_end;

    ieee80211_ratectl_init(vap);
    /* Complete setup. */
    ieee80211_vap_attach(vap, ieee80211_media_change,
        ieee80211_media_status, mac);
    ic->ic_opmode = opmode;

    return vap;
}

```

The `iwx_vap_create` performs some housekeeping to manage memory and establishes callbacks to be used by the net80211 system. For `iwx`, it establishes per-driver state (the `IWX_DEFAULT_MACID` and `IWX_DEFAULT_COLOR` values), which is used to coordinate with firmware about which station we use as a default.

For some functions that `iw_x_vap_create` hooks into, we retain the default method and intercept calls to it. For instance, we override the `iv_newstate` callback and filter it through `iw_x_newstate`.

The firmware for `iw_x` manages a lot of state itself; one example is probing, where the hardware can be asked to send probes for networks across supported channels, and we aren't able to send these packets directly ourselves.

The `iw_x` driver must hook the newstate methods to make requests to the firmware, updating its state machine. In this way, the net80211 and firmware state machines are kept in sync with host-level changes.

Sending packets

We have now covered enough of the driver that we can bring it up with `ifconfig` and ask the operating system to start sending packets.

When we are testing an interface, we might go through the following flow using `ifconfig`:

```
# ifconfig wlan0 ssid open-network up
```

These commands instruct `ifconfig` to bring up the interface and request that the net80211 stack join the open WiFi network `open-network`. It sets an address for the interface, but this doesn't lead to any packets on the wire (well, air).

Let's see what driver methods this series of commands translates into.

In our attach hook, we established two callbacks for the net80211 layer to use when it needs to send a packet: `ic_transmit` and `ic_raw_transmit`, and one to control the state of the interface (`ic_parent`).

```
ic->ic_raw_xmit = iw_x_raw_xmit;
...
ic->ic_parent = iw_x_parent;
ic->ic_transmit = iw_x_transmit;
```

The `up` part of the `ifconfig` command eventually calls the `ic_parent` callback. For `iw_x`, this is `iw_x_parent`:

```
static void
iw_x_parent(struct ieee80211com *ic)
{
    struct iw_x_softc *sc = ic->ic_softc;
    IW_X_LOCK(sc);

    if (sc->sc_flags & IW_X_FLAG_HW_INITED) {
        iw_x_stop(sc);
        sc->sc_flags &= ~IW_X_FLAG_HW_INITED;
    } else {
        iw_x_init(sc);
        ieee80211_start_all(ic);
    }
    IW_X_UNLOCK(sc);
}
```


`iw_xparent` directly controls the hardware, calling a function to tear down all hardware state if we are running `iw_xstop`, or if we aren't running yet, asking the hardware to be initially configured with `iw_xinit`. Once the hardware is ready, we then notify the net80211 stack that we are prepared to start with `ieee80211_start_all`.

The seemingly simple `ifconfig` action `up` results in a lot of hardware state being modified with the `iw_x` driver. This contributes partially to "bringing the interface up and down" being a suggested magic fix to resolve network inconsistencies.

The second part of the `ifconfig` command results in the net80211 stack taking quite a few steps. By passing `ssid open-network` to `ifconfig`, we are asking the net80211 subsystem to discover and join a network called `open-network`.

The IEEE 802.11 process to join a network is made up of several steps:

- probe for a network
- authenticate to the network
- associate with the network

Each of these steps requires a device to send management frames. First, we need to discover the network we want to join; networks regularly beacon their presence (this is what fills the network list in your menu bar). This gives the operating system a list of networks to try. When a device wants to join a network, it sends out a probe request for the target network and waits for a probe response. This process facilitates the transfer of configuration parameters between the network and the host, indicating to the host that the network is truly available.

The next step involves authentication to the network, followed by association. At this point, we move into the `RUN` state and can start using the wireless interface like any other network device.

As the stack moves between each state, it triggers a call to the `iv_newstate` function, which for `iw_x` is first intercepted by `iw_x_newstate`. This allows the driver to control the sending of packets for state transitions. We need this in `iw_x` because some of these transitions are handled by the device firmware rather than through direct packet transmission.

Rather than sending out probe requests directly, there is a firmware interface to trigger a scan of available networks. Once we have discovered the network and want to join it, we send a message to the firmware to add a station rather than sending out packets from the net80211 stack.

Not all management frames are sent by the firmware via an abstraction, and in those cases, the `iw_x_raw_xmit` callback is used by the system. If you are debugging a driver and wondering why the transmit path isn't always hit, it could be management frames exiting the raw path.

Conclusion

In this article, we have looked at how a driver probes, attaches, and sends some first packets. By using an existing driver, we can cover a lot of ground in the driver quite quickly.

.....

By passing `ssid open-network` to `ifconfig`, we are asking the net80211 subsystem to discover and join a network called `open-network`.

However, if you read through, you will see that `if_iwx.c` is a whopping 10,000 lines of code. That is more than we can address here.

This article, which has started to dig into hacking, has also glossed over many details. To join a network, we need to be able to both send and receive packets from the network interface.

If we don't get any packets, can we debug? What is offered by the system?

In Part 3 of the series, we will cover the built-in debugging features of the net80211 stack and how they hook into a driver for developer, testing, and troubleshooting.

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



The FreeBSD Project is looking for

- Programmers • Testers
- Researchers • Tech writers
- Anyone who wants to get involved

Find out more by

Checking out our website
freebsd.org/projects/newbies.html

Downloading the Software
freebsd.org/where.html

We're a welcoming community looking for people like you to help continue developing this robust operating system. Join us!

Already involved?

Don't forget to check out the latest grant opportunities at freebsd.foundation.org

Help Create the Future. Join the FreeBSD Project!

FreeBSD is internationally recognized as an innovative leader in providing a high-performance, secure, and stable operating system.

Not only is FreeBSD easy to install, but it runs a huge number of applications, offers powerful solutions, and cutting edge features. The best part? It's FREE of charge and comes with full source code.

Did you know that working with a mature, open source project is an excellent way to gain new skills, network with other professionals, and differentiate yourself in a competitive job market? Don't miss this opportunity to work with a diverse and committed community bringing about a better world powered by FreeBSD.

The FreeBSD Community is proudly supported by



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.

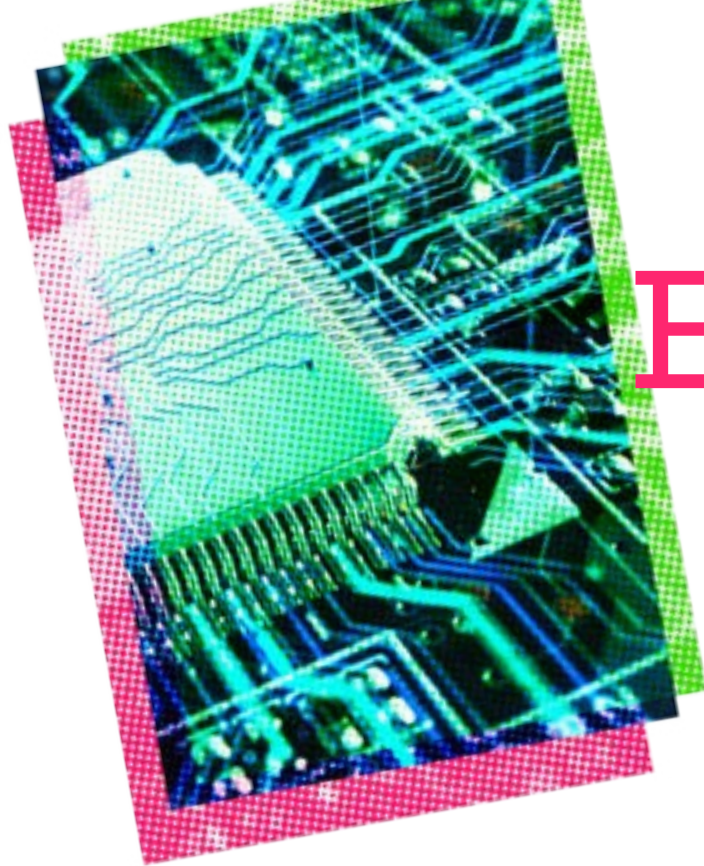
Please consider making a gift to support FreeBSD for the coming year. It's only with your help that we can continue and increase our support to make FreeBSD the high-performance, secure, and reliable OS you know and love!

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.
freebsd.foundation.org/donate





Embedded FreeBSD

Embedded FreeBSD: Looking Back and Forward

BY CHRISTOPHER R. BOWMAN

We've covered quite a lot of ground over the last year. While I hazard to guess that most people run FreeBSD on conventional AMD64-based PCs, we examined one of the embedded boards that FreeBSD works on: the [Digilent Arty Z7-20](#). While not inexpensive, the Arty Z7 provides an FPGA fabric connected to the CPUs, which differentiates it from less expensive boards like the Raspberry PI or Beagle Boards.

We began by discussing how to obtain a pre-built image for the board and how to communicate with it over a serial port. In the following article, we discuss rolling our own images and how to use the FreeBSD cross build infrastructure for the ARMv7 system on the Arty board. This vastly speeds development time. We also discussed how to customize the FreeBSD build and then load it onto an SD card, allowing us to create our own custom images.

Having learned to build and customize our own images, we learned how to set up a bhyve instance to run the AMD/Xilinx FPGA software so that we could experiment with FPGA fabric circuits.

Once we had a Linux instance running, we looked at the basic process for building circuits and getting them into the FPGA fabric. There were a lot of details to look at here. We had to create our circuits in a completely new language for hardware design called Verilog. We had to learn how to use the AMD/Xilinx tools to connect our circuits to pins on our chip, which were then connected to LEDs on our board. There was a repo that pulled all this together so that we could use our Linux bhyve instance to build our circuit. Finally, we learned two ways to load the circuit into our chip: one before the system boots and the other from within FreeBSD. We then saw glorious blinking LEDs, reminiscent of lights on a Christmas tree.

Having gotten our first circuit to work, we started exploring more complicated hardware where the CPU and our fabric circuit could communicate. To do this, we used the FreeBSD GPIO system, which required us first to figure out why the GPIO system wasn't working in our initial image builds. We briefly examined the probing of the GPIO driver and discovered that it was absent from our system because the hardware wasn't described in our Device Tree Binary (DTB). This led us to a brief discussion of Flattened Device Tree (FDT) files and how they describe the hardware of many embedded boards. We learned how to modify our

We examined one of the embedded boards that FreeBSD works on: the Digilent Arty Z7-20.

FDT file and build a DTB from it using the Device Tree Compiler (DTC). We learned how to get the FreeBSD loader to load our customized DTB before booting the kernel. Finally, once we had gone through all that, we were able to call the GPIO system from userspace to toggle external pins and, again, light up our LEDs.

In the most recent article, things got interesting. We looked at one of the many available PMOD modules, a dual seven-segment display. We built hardware in the FPGA fabric that could display values on the two displays and presented a register interface over the AXI bus to the CPU. We wrote entries for our FDT, describing the register interface to our hardware, and developed a driver to control the values on the seven-segment displays. In the end, we used the Unix **sysctl** framework as an API for the user space to set the seven-segment displays.

We've now reached a point where we can design circuits in Verilog and put them in the FPGA fabric of our Zynq chips. We've learned to build register interfaces that communicate over the AXI bus so that our CPU can easily interface with our custom hardware. We've learned to describe that hardware to the kernel and to build drivers that allow the FreeBSD system to interact with our hardware. We also learned how to interface with that hardware from the user space. So, what's next?

Once we had the basic capability to build circuits and put them in the FPGA fabric, we started learning ways to communicate between our hardware and the CPU subsystem of our Zynq chip. This opens a vast space for exploration and implementation, but there are some limitations. One of those limitations is bandwidth and concurrency. While extremely powerful and flexible, a register interface to hardware is bandwidth-limited. The CPU can only write to the registers so fast, especially when it needs to perform other tasks. Currently, our hardware is bandwidth-limited. It was great for the seven-segment displays, but if we wanted something more bandwidth-intensive, it wouldn't suffice.

Think about a video display. Our Arty board contains an HDMI output port. While a register interface might be viable for a character display, it wouldn't cut it for bit-mapped graphics. A 24-bit color depth 1280x720x60Hz display requires about 166 MB of data per second. We don't want to try to provide that via a register interface. For bit-mapped graphics, the conventional approach is to dedicate a chunk of memory into which the CPU can write and from which the display hardware can read. We need to explore how to build hardware to fetch (or store) data in main memory without using the CPU to move the data. We can utilize our register interface knowledge to enable a processor to configure parameters, such as the base address. However, we prefer our hardware handle the specifics of fetching the display buffer from main memory 60 times a second without CPU intervention.

Adding the capability for the CPU to describe objects in memory to which the hardware should read and write opens a whole new set of possibilities for our Zynq system designs. It also makes me wonder what the impact of that kind of bandwidth competition will be on our dual-processor Arm Cortex A9 system. Digilent makes another Zynq-based board that is like our Arty Z7-20. The [Digilent Zybo Z7](#) is more expensive than the Arty Z7, priced

We've now reached
a point where we can
design circuits in Verilog
and put them in
the FPGA fabric of
our Zynq chips.

at \$399 for the dual processor version compared to \$249 for the Arty. However, the Zybo's memory bus is twice as wide as the Arty's, operating at nearly the same frequency. Further, the Zybo offers 6 PMOD interfaces in comparison to the Arty's two. However, you'll lose the Arduino shell pinout. I think I'm more interested in the PMOD ports. Otherwise, both boards are based on the same chip. There shouldn't be any new drivers that need to be written. The FDT should remain essentially unchanged; it would be interesting to investigate the necessary changes to run this board.

Other things that might be interesting to investigate would be new PMOD modules. You can find a whole slew for sale on the [Digilent site](#). We used the PMOD SSD: Seven-Segment Display in an earlier article. Digilent has retired the [PMOD GPS](#), but I bought one before they did. It uses a UART interface, which, conveniently, is an onboard peripheral in our Zynq chip that can be connected to external pins via the fabric. It should be straightforward to connect this to the system, and I suspect there's open-source software that can communicate with this device via the UART link, enabling various GPS functions such as position and time. What I find interesting about this device is that it also provides a Pulse Per Second (PPS) output. I know that Poul-Henning Kamp has done some work with FPGAs and time-keeping in the past, and I would like to see how that is applicable here.

We haven't done any work with interrupts so far. Still, it should be relatively easy for a fabric circuit to generate an interrupt to the processor and then keep a register with the number of clock cycles between the PPS and the current time. When interrupt servicing software is scheduled, it could read this register and account for the latency between the interrupt and when the driver or time software runs. This might be useful to NTP software, I really have no idea, but it's something I'm curious about. It might be nice to have a local GPS synchronized stratum 1 time server.

I've got a variety of PMOD modules, including accelerometers, OLED displays, and LCDs. It might be interesting to interface with some of them. For example, if you ran an NTP server on your board (perhaps using hardware described above to improve accuracy), you could use an LCD to display atomic time and location continuously.

It almost slipped my mind, but the Zynq boards have Analog to Digital Converters (ADCs) built in. This would undoubtedly be an interesting area for exploration. Still, it may require some external analog circuitry for signal conditioning or buffering before passing on to the FPGA, and this might be outside the scope of an article in the *FreeBSD Journal*. It might be interesting to look at what is required to interface to these on-chip peripherals.

Of course, you can build your own hardware and easily interface it to the pins of the FPGA. I'm curious to hear what you would build if you could.

One obvious place I was going to explore that I hadn't mentioned is running Vivado under FreeBSD. If you've looked at some of my repos, you may have noticed that there is some experimental support for running Vivado under FreeBSD. However, it seems like someone has beaten me to the punch. Michał Kruszewski has written a detailed [blog post](#) on the topic. For most of what I do, this is perfect; I can build and simulate my circuits.

Other things that might be interesting to investigate would be new PMOD modules.

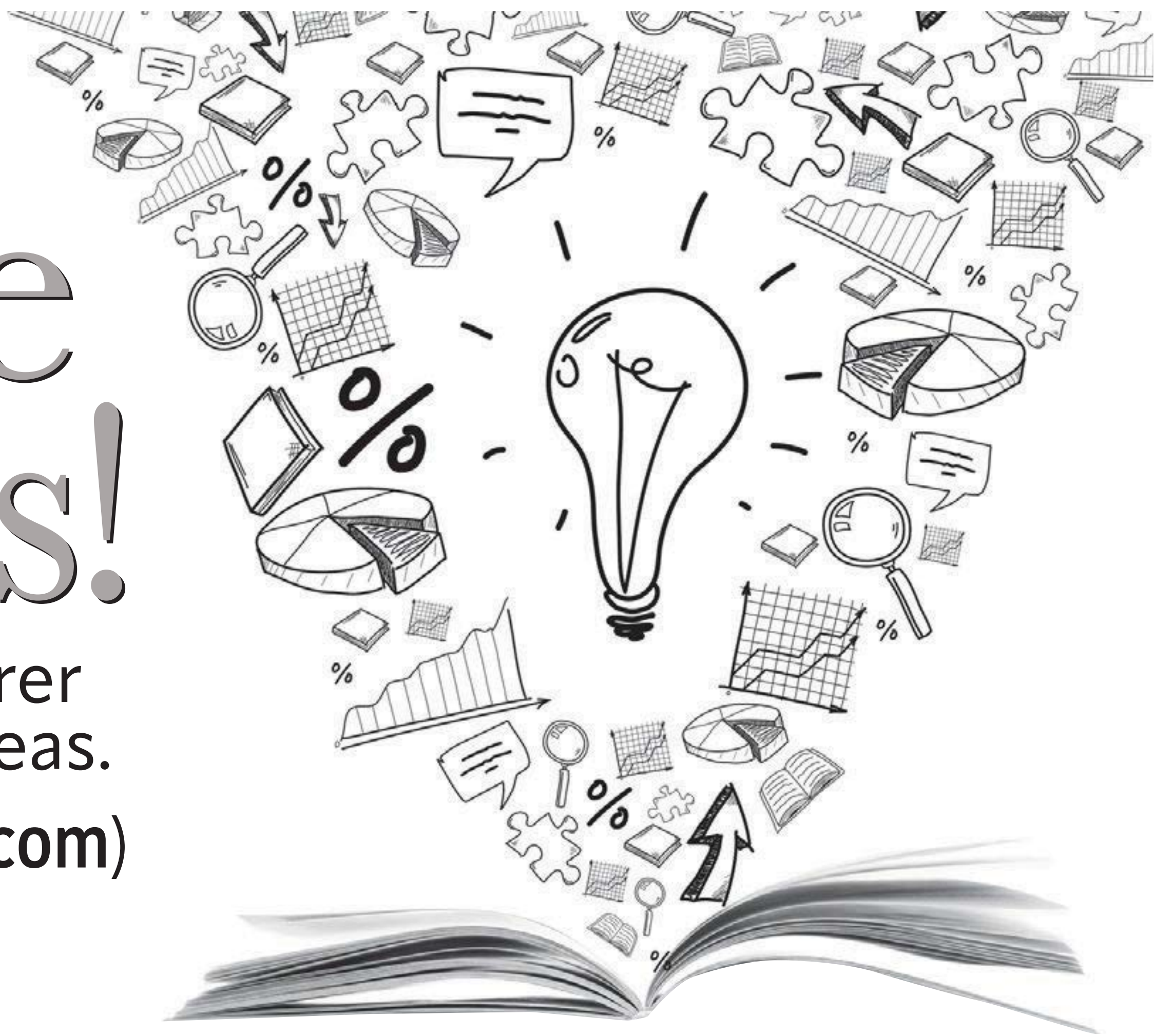
Things that aren't quite there yet are loading bitstreams from my FreeBSD host system and using the Vivado Logic Analyzer. The latter two don't work in my behyve Linux instance either, but perhaps I'll experiment with pass-through when FreeBSD 15.0 is released.

I hope you've found these columns useful. I'd appreciate your comments or feedback. 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.

Write For Us!

Contact Jim Maurer with your article ideas.
(maurer.jim@gmail.com)



BSDCan 2025

BY CHRISTOS MARGIOLIS

This year, I gave a talk at [BSDCan 2025](#) titled "[Vox FreeBSD: How sound\(4\) works](#)".

I arrived in Montréal on June 9, two days before the conference, but did not do much that day except rest.

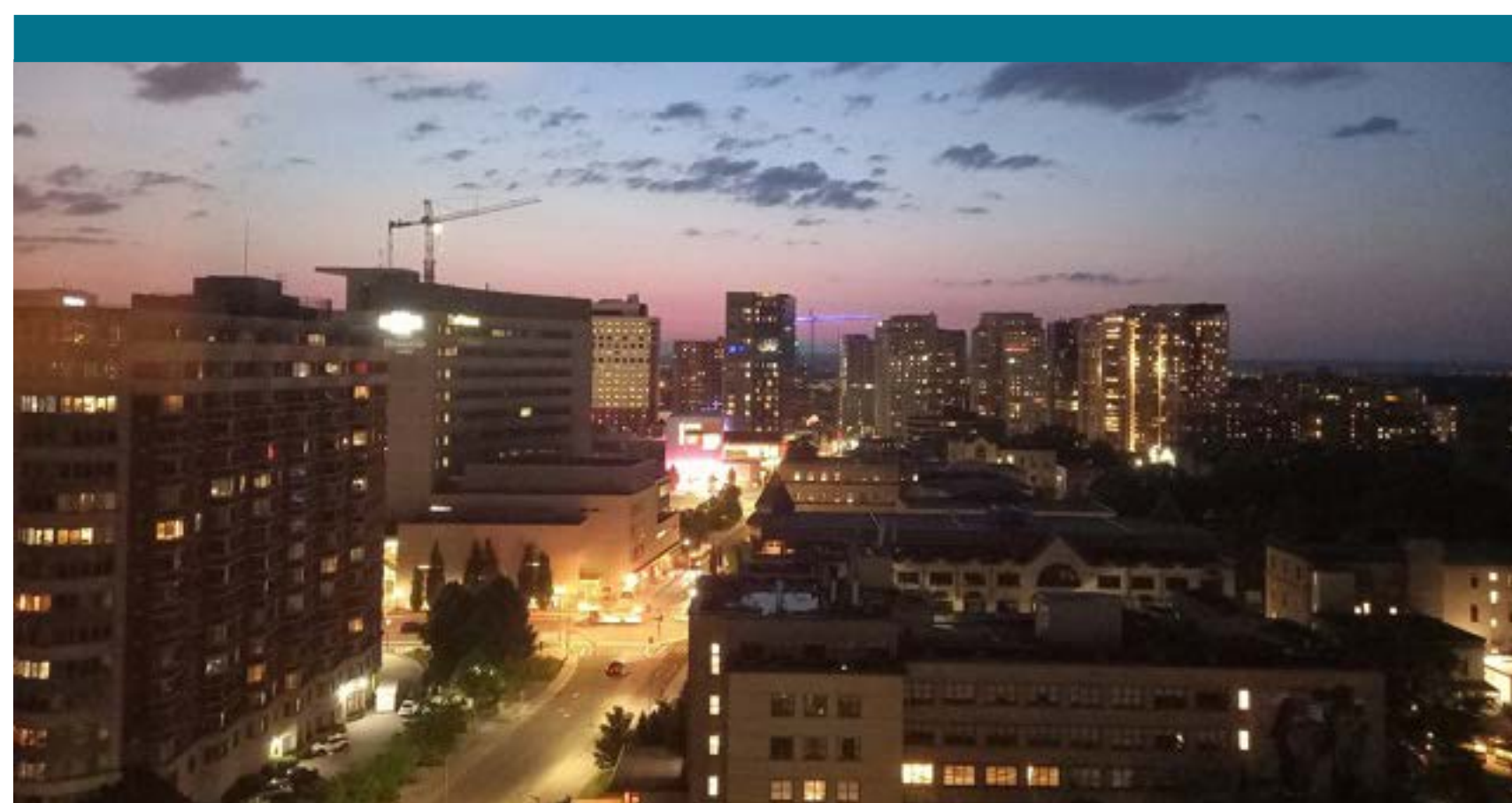
The next day, June 10, I took the bus from Montréal to Ottawa, where the conference is held. As soon as I got off the bus, I randomly stumbled upon Olivier Certner (olce@), who was walking by, and we went for lunch at a restaurant near the University of Ottawa — the venue and speaker residence — before checking in. In the evening, I met with Mateusz Piotrowski (Omp@), Bojan Novković (bnovkov@), and Kyle Evans (kevans@) for beers and dinner at Father & Sons Restaurant. In this common meeting place, we also met with other people from the conference.

The first two days of the conference (June 11 and 12), for me, were spent at the [FreeBSD DevSummit](#).

A highlight of the first day was the public discussion initiated by the Core Team, regarding AI usage on FreeBSD, which also continued during the breaks. The Core Team seemed to focus on the licensing concerns of mostly AI-generated code. My position, however, was that we should be *hardline against any use of AI*, primarily because of ethical and quality concerns. I find the licensing aspect of the AI question to be secondary in nature and relatively unimportant when compared with questions such as: Do we want to actively participate in possibly making the world a worse place? What kind of people will we attract to the project if we adopt a relaxed AI policy? How will this affect the quality of the project long-term? Do we really *need* AI and the complexity that comes with it? If licensing weren't a problem, would there be no problem in using AI? And many more questions... The good thing was that many people (some hesitantly) seemed to support my opinion.

I also had discussions, both technical and non-technical, with Mark Johnston (markj@), Joseph Mingrone (jrm@), Bojan, and Charlie Li (vishwin@), the latter of whom I did not know was also interested in FreeBSD audio/music production, and actually does DJ sets using it.

After the first day of the DevSummit, we headed for pizza at the university residence, but I retired to my room relatively early to do some work and get my slides done.



The second day of the DevSummit, June 12, started with a talk from AlphaOmega on security auditing, followed by a discussion about [FreeBSD 15.0 technical planning](#), including PkgBase. After lunch break, Brooks Davis (brooks@) gave a fascinating talk — probably my favorite technical one throughout the DevSummit — on upstreaming the [CheriBSD](#) branch. Even though the talk was about upstreaming, he also took the time to give an excellent overview of what CHERI, CheriBSD, and capabilities are, and how the built-in memory safety feature manages to catch various bugs in the FreeBSD code. Then, the [FreeBSD Foundation](#) gave an update on recent work and funding, including the [Laptop Support and Usability Project](#), which I am a part of. Speaking of laptops, towards the end of the DevSummit, I spent some time with Alexander Ziaee (ziaee@), who has been working hard on improving documentation, and attempted to debug a sound issue on his laptop. In the evening, I met with Benedict Reuschling (bcr@) and a few other people and went for dinner at a nice seafood restaurant.

BSDCan took place on June 13 and 14. It started with a keynote speech from Margot Seltzer, a famous computer scientist, about [Hardware Support for Memory-Hungry Applications](#).

I attended many talks during those two days, but a few that stood out for me include:

- [ShengYi Hung: ABI stability in FreeBSD](#). He showcased his new experimental tool, which detects ABI changes based on differences between CTF data. Then followed an interesting discussion, including me, Mark Johnston, John Baldwin (jhb@), Warner Losh (imp@), and the speaker, about the limitations and oversights of this tool, and how to use it in real-world scenarios.
- [Marshall Kirk McKusick: A History of the BSD Daemon](#), as well as an update on the upcoming third edition of *The Design and Implementation of the FreeBSD Operating System*. I always enjoy Kirk's presentation style.
- [Bojan Novković: Hardware-accelerated program tracing on FreeBSD](#). He presented his recent work on the hwt(8) framework.
- [Zhuo Ying Jiang Li: Improvements to FreeBSD KASAN](#). She gave an overview of KASAN, FreeBSD's kernel address sanitizer. She presented her work on it as part of her involvement with CheriBSD.

A few talks I would like to have attended, but didn't manage to:

- [John Baldwin: ELF Nightmares: GOTs, PLTs, and Relocations Oh My](#).
- [Andrew Hewus Fresh: The state of 3D-printing from OpenBSD](#).
- [Hans-Jörg Höxer: Confidential Computing with OpenBSD — The Next Step](#).
- [Andreas Kirchner, Benedict Reuschling: Enhancing Unix Education through Chaos Engineering and Gamification using FreeBSD](#).

I gave my talk on June 14, the last day of BSDCan. It sparked lots of questions and conversations, which continued even after the talk. Apparently, there are more people than I thought who would really like to be able to use FreeBSD for music and audio production

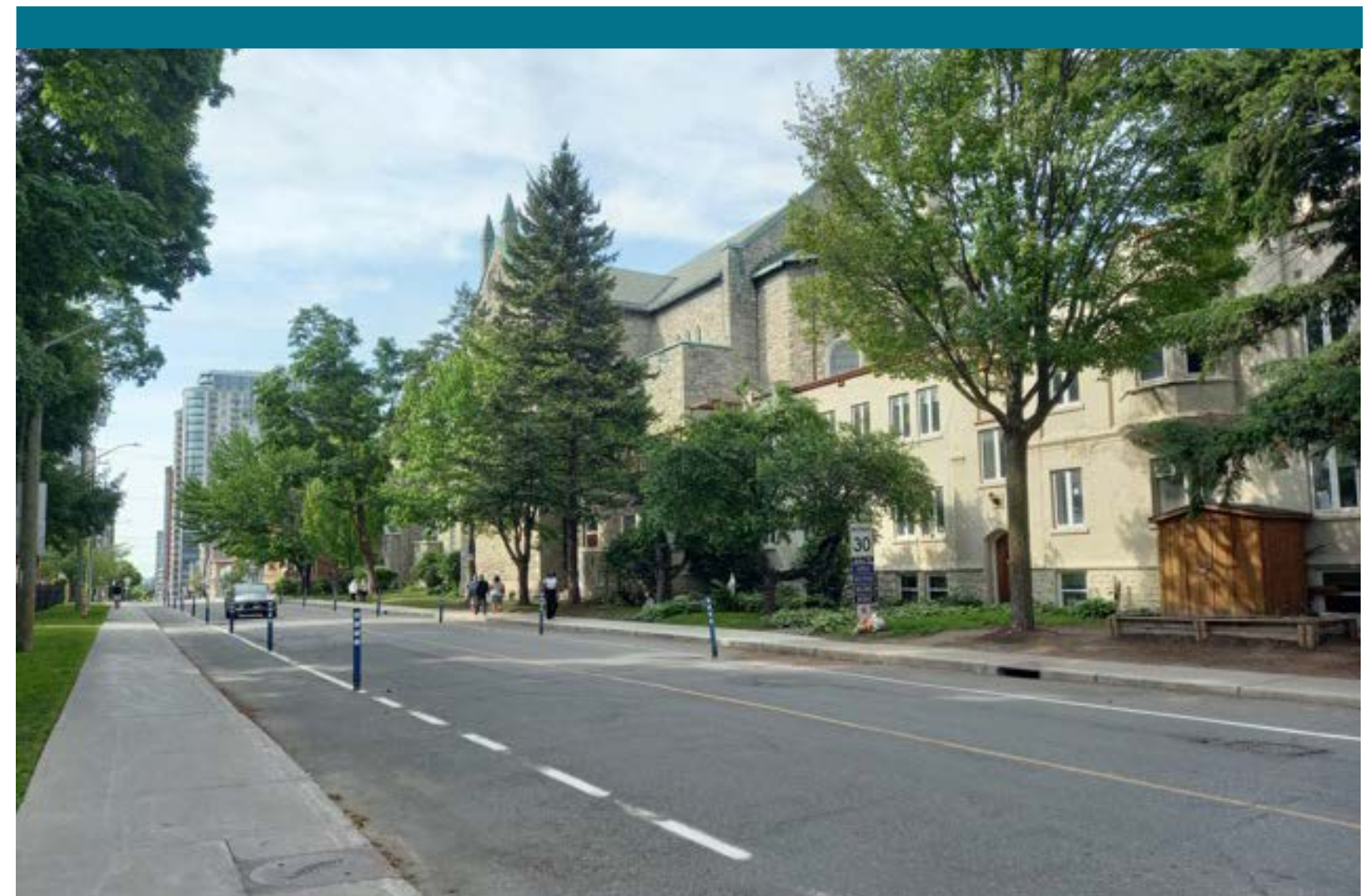
A highlight of the first day was the public discussion initiated by the Core Team, regarding AI usage on FreeBSD.

or in large audio installations, so the talk seemed to have inspired some of them, which is excellent.

After the conference's Closing Session, we headed to a nearby market square for the social event. I spent most of my time with Mark Johnston, Andreas Kirchner, and Mateusz Piotrowski, and had many interesting conversations.

The next day, I went back to Montréal to enjoy some days off before my return home.

As always, conferences are great opportunities to make up for the solitary nature of programming and meet with the people behind the screens with whom we exchange emails every day. Apart from the fact that we got work done and exchanged interesting technical ideas, what I enjoyed even more was the unexpected, deep discussions I had with people, including people I had never met before.



CHRISTOS MARGIOLIS is an independent contractor and FreeBSD src committer from Greece.



The FreeBSD Project is looking for

- Programmers • Testers
- Researchers • Tech writers
- Anyone who wants to get involved

Find out more by

Checking out our website

freebsd.org/projects/newbies.html

Downloading the Software

freebsd.org/where.html

We're a welcoming community looking for people like you to help continue developing this robust operating system. Join us!

Already involved?

Don't forget to check out the latest grant opportunities at freebsd.foundation.org

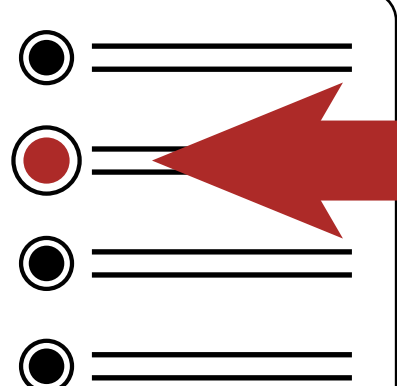
Help Create the Future. Join the FreeBSD Project!

FreeBSD is internationally recognized as an innovative leader in providing a high-performance, secure, and stable operating system.

Not only is FreeBSD easy to install, but it runs a huge number of applications, offers powerful solutions, and cutting edge features. The best part? It's FREE of charge and comes with full source code.

Did you know that working with a mature, open source project is an excellent way to gain new skills, network with other professionals, and differentiate yourself in a competitive job market? Don't miss this opportunity to work with a diverse and committed community bringing about a better world powered by FreeBSD.

The FreeBSD Community is proudly supported by



Contents



Events Calendar

BSD Events taking place through November 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.



OpenZFS User and Developer Summit 2025

October 25-28, 2025

Portland, OR

https://openzfs.org/wiki/OpenZFS_Developer_Summit_2025

This year, the User Summit will explore a wide range of topics designed to support and connect the OpenZFS community.



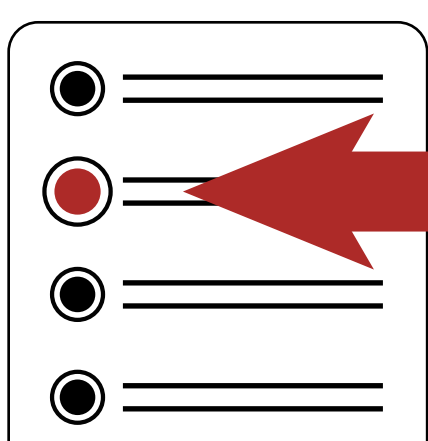
November 2025 FreeBSD Vendor Summit

November 6-7, 2025

San Jose, CA

<https://freebsdoundation.org/news-and-events/event-calendar/fall-2025-freebsd-summit/>

The Summit provides commercial FreeBSD users with the unique opportunity to meet face-to-face with developers and contributors to get features requested, problems solved, and needs met. It also opens up discussion on improving and enhancing the operating system.



Contents