October/November/December 2025

FreeBSD® JOURNAL

Topic: FreeBSD 15.0

FreeBSD 15.0: Fixes and Features

Universal Flash Storage on FreeBSD

VOX FreeBSD: How Sound Works

Credentials Transitions with mdo(1) and mac_do(4)

Printf(“Hello, srcmgr\n”);

FreeBSD and Google Summer of Code 2025

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!

2026 Editorial Calendar

- Jan/Feb/March **Laptop/Desktop**
- April/May/June **Improving Software Quality**
- July/August/Sept **Production Deployments**
- Oct/Nov/Dec **to come**

Find out more at: freebsdfoundation/journal

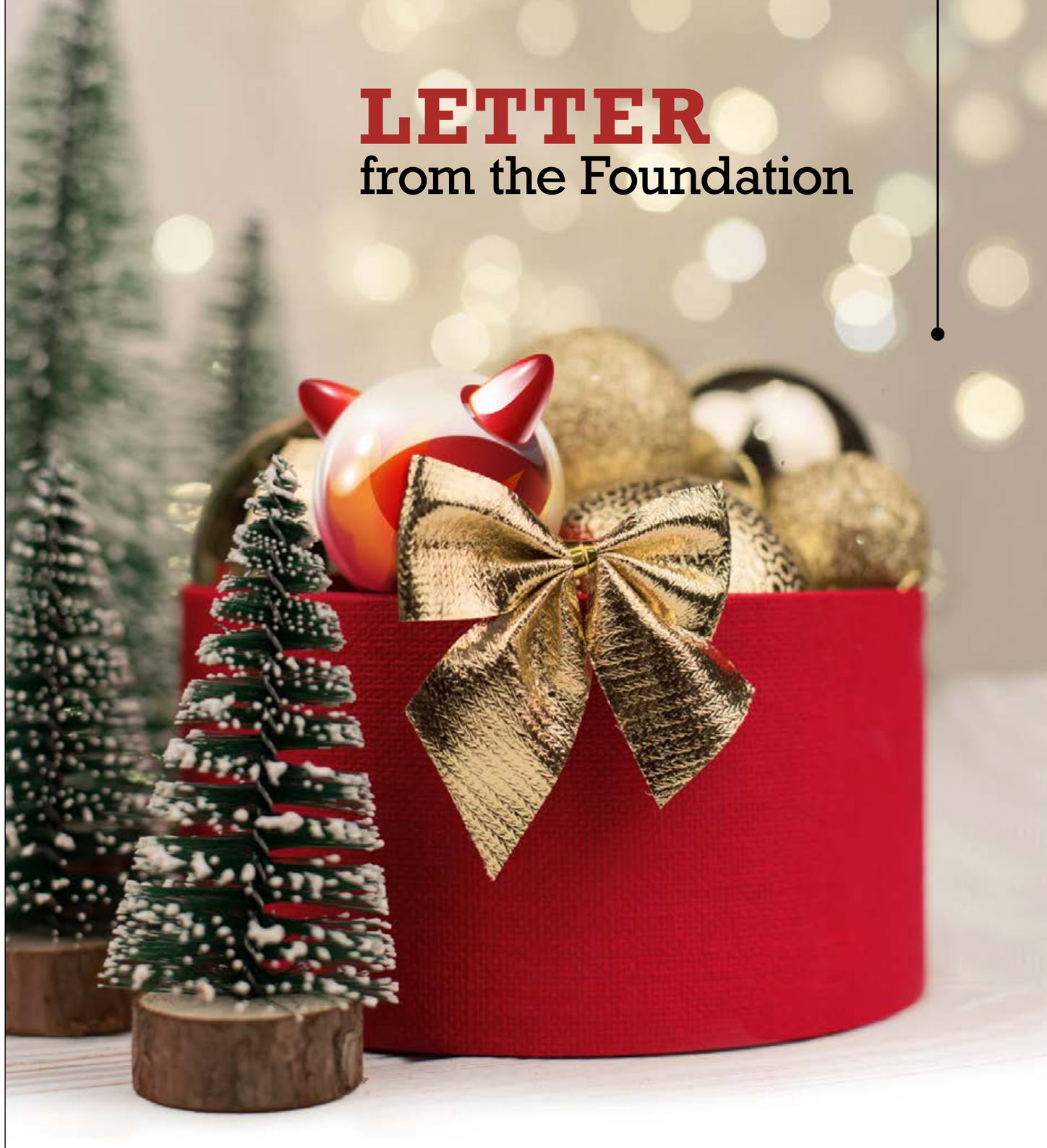
FreeBSD® JOURNAL

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

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

Design & Production • Reuter & Associates

FreeBSD Journal (ISBN: 978-0-61 5-88479-0) is published 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@freebsdfoundation.org



LETTER from the Foundation

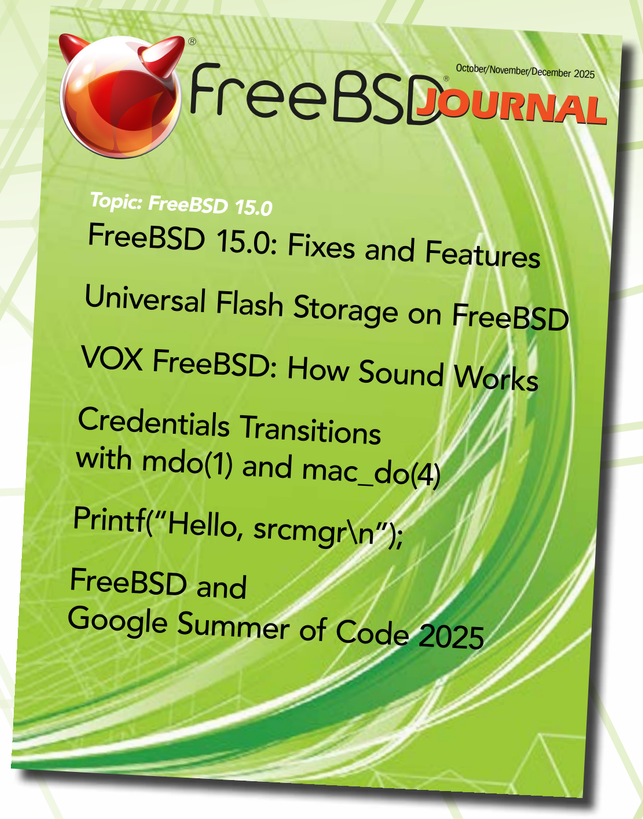
Thank you for spending time with the *FreeBSD Journal* this year. We're grateful for our readers, as well as the authors, editors, and volunteers who bring each issue to life. Your passion and effort keep the *Journal* going and help share the work happening across the *Project*. Wishing you a happy, restful holiday season from all of us at the *FreeBSD Foundation*.

Deb Goodkin
Executive Director
FreeBSD Foundation

October/November/December 2025

FreeBSD® JOURNAL

Topic: FreeBSD 15.0



WeGet letters

by Michael W Lucas

Dear Why Do They Let You Write This Column,

FreeBSD 15 is now out, and it includes the base system as packages. In the January/February 2022 issue of the *FreeBSD Journal*, you spent your entire column ranting about pkgbase. You were wrong. Obviously wrong. Is that enough to finally make you shut up and quit answering our letters?

—Wants Useful Answers

Dear Wanna-Play-Gotcha,

Getting me to shut up is a solved problem. It costs $200 per hour (https://givebutter.com/c/Penguicon24/auction/items/258146). And why do they let me write this? Because John Baldwin, *FreeBSD Journal* editor, was desperate for a letters column. Now that he has one, he's even more desperate, but for completely different reasons. A change is as good as a rest.

Now on to your real question. Packaged base. What's up? Why?

I did not spend the "entire column ranting about pkgbase." Yes, I said the only true way to upgrade BSD was to build it from source and install it yourself. I also said that packaged base was "the dread dragon of FreeBSD, devouring every developer who sets out to conquer it."

Dragons fill vital ecological space. They're warnings. They're limits: *You shall go this far, and no further.* We call them evil only because people hate limits. When we encounter a limit or a dragon, we try to slay it. We still remember Saint George slaying an innocent dragon to save a princess. We remember because royalty keeps spreading the tale in the hope that we'll rescue them from the next dragon, all while neglecting to remind us that the king thought that letting dragons devour people was regrettable but acceptable until his own daughter landed on the menu.

Intelligence has very little connection to the real world and provides zero protection against peer pressure.

Anyway. Slay the dragon, and you'll go down in history.

The real problem here? Developers, like most "technical" people, are quite intelligent. Intelligence has very little connection to the real world and provides zero protection against peer pressure.

"Self-hosting," the ability to build a system from code and tools on the system, was integral to primordial BSD and remains critical today. A BSD system should ship with all the source code and tools needed to rebuild itself. The natural state of a BSD system is a single, whole entity. Breaking it up into pieces is like selling apples by the slice. Packaging sure *feels* convenient, though. The cool penguin kids have base system packaging! I surrendered any hope of being cool shortly after being born, so peer pressure triggers mere ennui.

Some folks found building FreeBSD from source code "slow" and "annoying." It might appear so to the unenlightened, but it's all about technical correctness. Correctness is a primary goal of every BSD, and watching the compiler churn as it recursively rebuilds the compiler for the eleventy-ninth time is a minor price to pay to achieve such pinnacles of purity. Technical correctness is not the worst kind of correctness. It merely feels that way.

In 2006, Colin Percival released freebsd-update(8) to allow the impatient to bypass this rite of passage, leading to wider adoption and deployment of FreeBSD.

FreeBSD also has the ports system. Ports allow the sysadmin to build add-on software exactly as desired and bundle it up in convenient tarballs so you can install them anywhere. The thirty-four thousand plus add-on programs in the ports collection supports a tangle of options and dependencies that nicely illustrates exactly why we build BSD as a monolithic whole, and the package management software needed to cope with all these interactions. The modern package management tools in pkg(8) were first included in FreeBSD 9.1 at the end of 2012.

In 2006, Colin Percival released freebsd-update(8) to allow the impatient to bypass this rite of passage, leading to wider adoption and deployment of FreeBSD.

No sooner had the new package tools been launched than folks obsessed with mere "usability" ignored the Unix philosophy of "many small tools that each handle one task" and started babbling about the convenience of using a single tool to manage both the base system and add-on software. How hard could it be? After all, bloated text editors like Emacs and vim are also built into a whole bunch of files. They could just tar up the whole base system, give it a packing list, and call it done?

Well, no.

Innumerable developers succumbed to the temptation to wrangle FreeBSD into friendly packages until a public call for tests went out in 2016. As you might expect, that's when everything went wrong. If you damaged your installation and wanted to remove all packages and start over, you would lose your operating system. That detail got promptly fixed, but it turned out there are dozens and dozens of edges and hundreds of interactions between them.

That original call for testing claimed that FreeBSD 11 would ship with a packaged base system. That slipped release after release, as lingering bugs were found and fixed.

Like all volunteer software projects, pkgbase hung at 99% complete for years.

FreeBSD prides itself on its open management, but one person in the project wields phenomenal cosmic power: the release engineer. The FreeBSD 15 release engineer put his

foot down and said, "We are not only packaging the base system for 15, I will delay the release until the base system is packaged." People flinched but got to work. The pkgbaseify tool converts FreeBSD 14 hosts to use pkgbase so they can be easily upgraded to FreeBSD 15 with the packaging tools.

After all this testing, pkgbase should work for most people. I have no doubt that somebody with an excess of cleverness will use it to render their hosts unbootable or install Rails as their new kernel, but users are responsible for their own ideas, and I have a big bucket of popcorn ready.

A clean install of FreeBSD 15 on my crashbox shows 311 packages, all but one with a name beginning with "FreeBSD." The exception is, of course, "pkg." Frankly, if you delete the FreeBSD-kernel-generic package on a production server, you deserve what you get, which is a warning that you can't delete the kernel. Or libc, or the linker, or any of the other obvious candidates. You can remove the compiler but, as I said in the original pkgbase discussion, "real operating systems ship with fully functional compilers in the default install." I also offered choice words for folks who suffer from a morbid fear of compilers.

A transition such as this merits a deeper look and a consideration of historical context. Nineteen years ago, Colin released freebsd-update out of a misguided impulse to simplify upgrading and patching. The release engineer who held FreeBSD 15 hostage until it included a packaged base system?

Colin Percival.

I think I see the *real* problem…

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

MICHAEL W LUCAS iis the author of *Absolute FreeBSD*, *Dear Abyss*, *SSH Mastery*, and too many other books. Learn more at https://mwl.io.

FreeBSD 15.0: Fixes and Features

BY JOHN BALDWIN

The FreeBSD community continues to push forward with the release of 15.0. This release includes numerous features, refinements, and bugfixes relative to 14.0, which was released in November of 2023. Highlights of some of the changes are listed below, but more details can be found in the release notes.

Improving Project Structure

While FreeBSD's developers have merged many patches over the last two years, they have also refactored several of the project's processes and structures. These changes aim to streamline development workflows and make optimal use of developers' time.

Colin Percival proposed several changes to FreeBSD's release schedule shortly after the release of 14.0. As he detailed in the journal earlier this year, the new schedule features a fixed cadence of both major and minor releases. 15.0 is the first major release following the new schedule.

At last year's BSDCan, FreeBSD's core team announced the new srcmgr team to manage the source repository. Delegating tasks such as src commit bits to this new team allows the core team to focus its efforts on strategic planning for the project as a whole.

15.0

This release includes numerous features, refinements, and bugfixes relative to 14.0.

Packaged Base System

The pkg(8) tool has proven itself as a mature system for managing binary packages. FreeBSD has been using pkg(8) to manage third-party packages built from the ports collection for many years. Over the past few years, a group of developers has worked to provide binary updates to the base system using pkg(8). This has included enhancements to the pkg(8) tool as well as changes to the base system build process to integrate with the pkg(8) tool. 15.0 will be the first major release supporting binary updates via pkg(8). The older system of distribution sets managed by freebsd-update(8) will also be supported in 15.x, allowing end users a graceful transition between these systems. FreeBSD's developers expect to switch the installer to use a packaged base system in the next major release.

Focusing Development Effort on Future Systems

Developer time and effort are scarce resources. To provide a high-quality system, FreeBSD has long focused on contemporary, widely deployed systems. Over the past major releases, FreeBSD has chosen to deprecate support for older CPU architectures that are seeing declining use in industry and limited developer support. 14.0 deprecated several 32-bit architectures that will not be supported as a standalone architecture in 15.0, such

as 32-bit x86 and 32-bit PowerPC. The 64-bit versions of both architectures will continue to support running 32-bit binaries in 15.0 and beyond. However, 32-bit kernels for these architectures are no longer supported in 15.0, and release artifacts such as install images will not be provided for 15.0.

Networking

15.0 includes support for new networking devices as well as improvements to TCP. Nvidia contributed changes to support inline IPsec offload, enabling smart NICs to offload IPsec encryption/decryption from the host CPUs to the NIC. This is similar to kernel TLS offload, but for IPsec. The mlx5en(4) driver supports IPsec offload on ConnectX-6 and later adapters. Local (UNIX domain) sockets were refactored in 15.0, resulting in increased throughput and reduced latency for local stream sockets.

15.0 includes support for new networking devices as well as improvements to TCP.

Storage

Several new storage features are included in the upcoming release. Samsung contributed a driver for the Universal Flash Storage standard, an alternative to the eMMC standard used for embedded flash storage. The driver's author, Jaeyoon Choi, covers this in more detail in "Universal Flash Storage on FreeBSD" in this issue. 15.0 also includes support for NVMe over Fabrics using the TCP transport as covered in a previous journal article. Since that article was published, support for NVMe-oF has been merged into the ctld(8) daemon, and the nvmfd daemon has been removed.

Also included in 15.0 is a native implementation of the inotify(2) family of system calls. This implementation is API-compatible with the same system calls as Linux and is available for both native FreeBSD binaries and Linux binaries running under the Linux compatibility layer. For many use cases, inotify(2) is both more reliable and more efficient than EVFILT_VNODE kernel events available via kevent(2). It is also a widely used API in existing desktop software such as KDE.

Virtualization

FreeBSD's type 2 hypervisor, bhyve, includes several updates in 15.0 as well. Both the in-kernel monitor and the userspace hypervisor are now supported for the 64-bit ARM and RISC-V architectures. A few advanced features, such as PCI pass-through, are not yet supported, but both FreeBSD and Linux guests using existing bhyve device models, such as VirtIO, are fully supported on both new architectures.

In addition to increased architecture support, bhyve can now use the net/libslirp package to provide a userspace backend for network devices. This allows the host to connect to guests over a network connection without requiring additional host network configuration, such as tap(4) devices.

Architecture-Specific

15.0 includes a processor tracing framework, hwt(4), which collects streams of events logged by CPUs. These events include details about software execution, such as control

flow changes, exceptions, and timing information. The framework supports events logged by ARM's Coresight and Statistical Profiling Extension (SPE) and Intel's Processor Trace (PT).

This release also includes support for AMD's IOMMU, which is particularly useful on systems with many cores. IOMMUs on x86 systems provide several features. The main purpose of an IOMMU is to provide an alternate address space for device DMA requests, which is useful both for virtualization (such as PCI pass-through) and for security (restricting memory access for untrusted devices). On x86, IOMMUs also interpose on interrupt delivery, permitting device interrupts to be routed to CPUs with numerically larger IDs. Previous releases of FreeBSD have included support for Intel's IOMMU (DMAR), and 15.0 introduces support for AMD's IOMMU.

Extended Error Reporting

Traditionally, in POSIX systems, system calls report errors during execution by returning an integer error code. This error code is available in the special global variable errno and can be translated to strings in functions such as strerror(3). 15.0 introduces a new extended error facility in the kernel, which saves additional information about an error, including an additional string description and the location in the source code of the error. The string description can be retrieved after a failed system call via the uexterr_gettext(3) function. The err(3), errx(3), warn(3), and warnx(3) family of functions will include the extended string description in the messages output to stderr automatically. Extended error information is also available via ktrace(1).

Third-Party Software

FreeBSD's base system includes several components that are externally maintained. As with every release, 15.0 updates many of these components by importing newer versions of the upstream software. The list of updates is too long to mention here, but a few deserve special mention. OpenZFS has been updated to the latest release, 2.4.0. OpenSSL has been upgraded to the current long-term support release (3.5), ensuring upstream support for the life of the stable/15 branch. The current version of MIT Kerberos has been imported into the base system, replacing the older Heimdal implementation. Toolchain utility programs such as ar(1) and size(1) are now provided by LLVM rather than the ELF toolchain project. This enables support for link-time optimization (LTO) in the base system toolchain.

Conclusion

FreeBSD 15.0 incorporates fixes and features contributed by a broad community over the last two years. Thank you to everyone who has contributed to this release by testing snapshots, reporting bugs, submitting patches, working with users on social media, and performing countless other tasks. We hope you enjoy FreeBSD 15.0. Please join us as we continue to move forward with FreeBSD 16 development!

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

Universal Flash Storage on FreeBSD

BY JAEYOON CHOI

Universal Flash Storage (UFS) is a high-performance, low-power storage interface designed for mobile, automotive, and embedded environments. Today, UFS is widely deployed and has become the successor to eMMC in most Android flagship smartphones. It also appears in some tablets, laptops, and automotive systems. Linux has supported UFS since around 2012, OpenBSD since 7.3, and Windows since Windows 10.

However, FreeBSD did not have a UFS driver. If we were to bring UFS's proven maturity from other ecosystems into the FreeBSD storage stack, FreeBSD would become a viable choice in many mobile and embedded domains. The question "Why doesn't FreeBSD have a UFS driver yet?" became my personal motivation, and this article describes the path I took to answer it.

Until last year, I had never used FreeBSD. Over the course of approximately six months, I studied FreeBSD, analyzed its codebase, and eventually implemented a UFS driver. I hope this case shows that even someone new to FreeBSD can develop a device driver with a systematic approach.

This article provides a brief introduction to UFS, explains its architecture, development process, and driver design, shares its current status and future plans, and finally presents a hands-on environment using QEMU, allowing readers to follow along.

The question "Why doesn't FreeBSD have a UFS driver yet?" became my personal motivation.

Note: FreeBSD also has the traditional UFS (Unix File System). In this article, "UFS" means Universal Flash Storage, not the file system. The driver name in the codebase is ufshci(4).

Universal Flash Storage Overview

In mobile storage, low latency and low power are essential, and compatibility with existing systems is also important. Rather than inventing an entirely new standard, UFS combines existing standards to achieve these goals. As a result, it was quickly adopted by the market. It retained the benefits of low power and reliability from its component standards, albeit at the cost of somewhat greater implementation complexity.

At the interconnect layer, UFS uses MIPI M-PHY (a reliable, differential high-speed serial interface) together with MIPI UniPro (a link layer with strong power management). On top of that, the transport protocol layer uses UTP, and the application layer uses a SCSI command subset whose reliability and compatibility are already well proven.

Where UFS is used

UFS may feel unfamiliar, but you are likely already using it. Most Android flagship smartphones use UFS for internal storage. UFS is also used in some low-power tablets and ultralight laptops, as well as in automotive infotainment, where reliability is critical.

Many high-performance ARM application processors integrate a UFS host controller, and some low-power x86 platforms (e.g., Intel N100) support a UFS host controller as well. Nintendo's recently released handheld console, Nintendo Switch 2, uses UFS for internal storage.

As UFS adoption continues to grow, adding native UFS support in FreeBSD broadens its applicability to mobile and embedded systems. Anticipating this demand, I started this project.

UFS Architecture

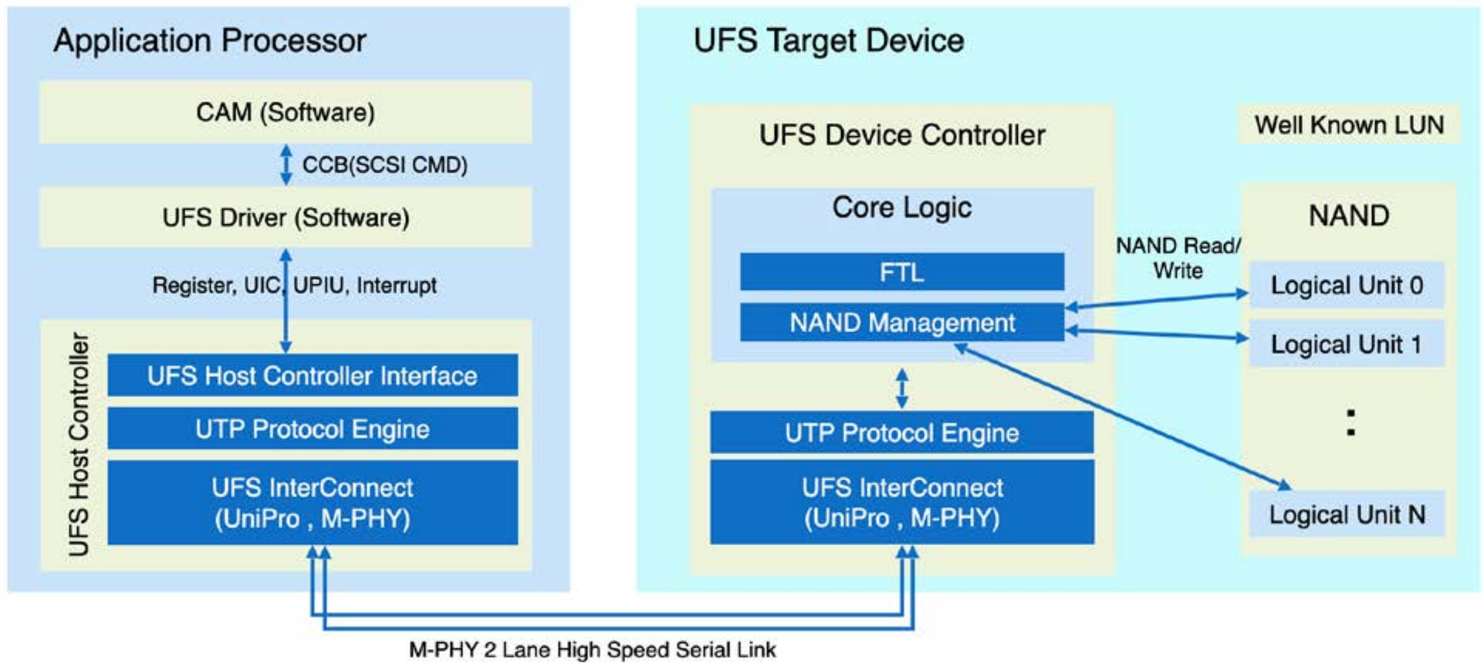


Figure 1. UFS System Model

A typical UFS system consists of a UFS target device (usually a BGA package on the PCB) and a UFS host controller integrated into the application processor SoC. The two components communicate over a high-speed serial link.

When an I/O request arrives, FreeBSD's CAM (Common Access Method) subsystem builds a SCSI command in a CCB (CAM Control Block) and passes it to the driver. The driver encapsulates the SCSI command in a UPIU (UFS Protocol Information Unit) and enqueues it on the UFS host controller's queue.

The host controller enqueues the request and rings the doorbell; data moves via DMA, and completions arrive via interrupts. The UFS device executes the SCSI subset command (e.g., READ/WRITE), accesses NAND flash, and returns completion or error status to the host controller.

In this way, the UFS device driver controls the host controller to perform reads and writes to the UFS device.

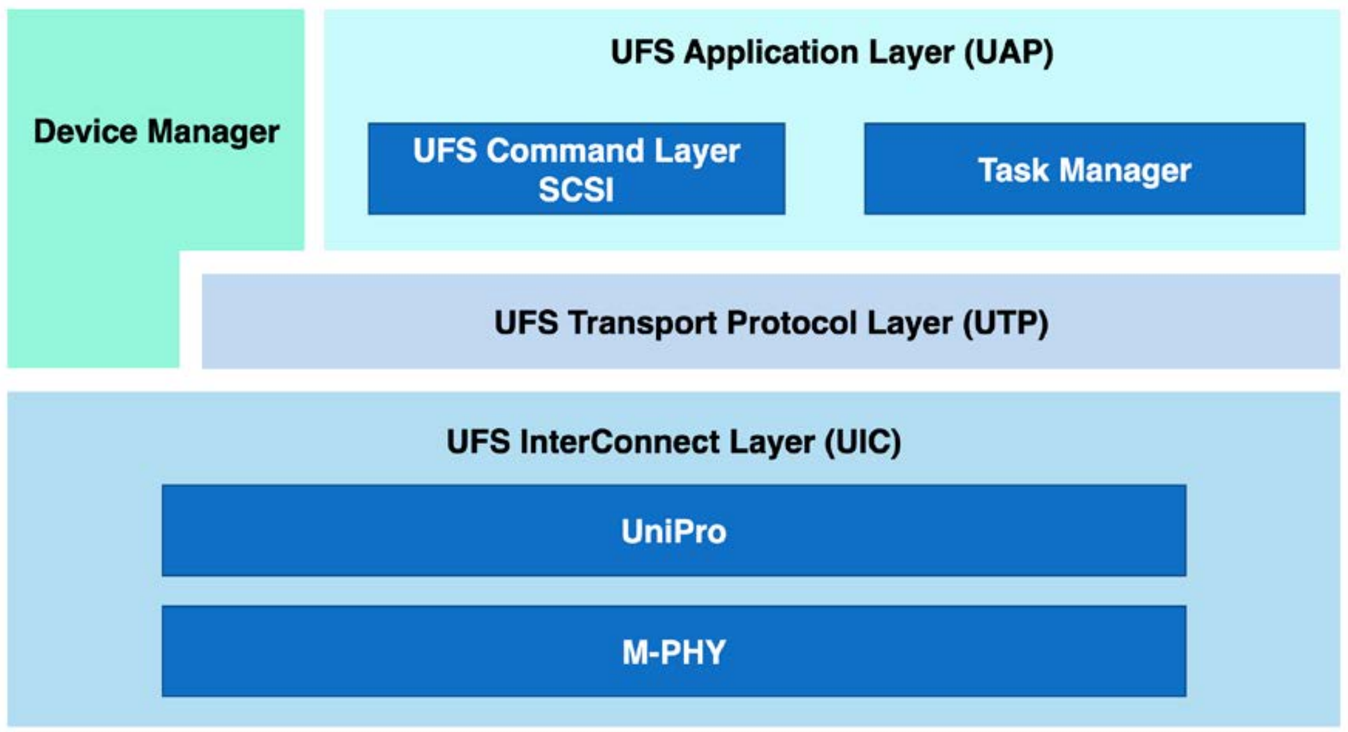


Figure 2. UFS Layered Architecture

As noted earlier, UFS layers several existing standards and is organized into three layers: interconnect, transport, and application. Their roles are:

- **UIC (UFS InterConnect Layer):** Manages the link. It performs Link Startup over M-PHY/UniPro, adjusts gear and lane settings to balance performance and power, supports power states (Active, Hibernate), detects errors, and performs recovery. The UFS driver controls this layer via registers and UIC commands.
- **UTP (UFS Transport Protocol Layer):** Transports admin and SCSI commands. The host controller maintains Admin and I/O queues; the driver enqueues requests and rings a doorbell. Data moves via DMA, and completions arrive via interrupts.
- **UAP (UFS Application Layer):** Handles commands (e.g., READ/WRITE) and command queue control. Although UFS defines multiple command sets, in practice, only the SCSI command subset is used. The UFS driver does not create SCSI commands; it encapsulates CAM-generated SCSI commands in UPIUs for UTP. This allows CAM's standard paths for scanning, error handling, and retries to be reused as-is. Through this layer, CAM treats UFS as a standard SCSI device.

UFS layers several existing standards and is organized into three layers: interconnect, transport, and application.

Key advantages (high performance, low power)

- **Performance:** Compared with eMMC's half-duplex, parallel interface, UFS's full-duplex, high-speed serial interface delivers higher bandwidth and lower latency. The submission path is lightweight, built around queues, DMA, and interrupts. As a result, performance is solid even with a single queue, and multi-circular queues (MCQ) improve scalability on multicore systems. WriteBooster further improves burst-write performance by using an SLC region of NAND.
- **Power efficiency:** The UIC link raises the gear only when I/O is active and quickly drops to a low-power state when idle. The standard defines power-state transitions, enabling longer battery life in thermal and power-constrained mobile form factors. In practice, tablets using UFS instead of NVMe have been reported to gain roughly 30-90 minutes of battery life.
- **Compatibility:** Because UFS uses a SCSI command subset, existing SCSI infrastructure can be reused. On FreeBSD, CAM handles SCSI processing, while the UFS driver encapsulates CAM-generated SCSI commands in UPIUs for UTP, making integration with CAM straightforward.

History and future of UFS

The UFS standard is published as JESD220 (UFS), and the host controller interface as JESD223 (UFSHCI). UFS/UFSHCI 1.0 was released in 2011. In 2015, UFS 2.1 devices first shipped in the Samsung Galaxy S6, marking the start of broad commercialization. UFS 3.0 improved link speed and introduced WriteBooster for burst writes. UFS 4.0 added multi-circular queues (MCQ), improving multicore scaling. UFS 4.1 is the current release, and UFS is deployed in most flagship smartphones.

As on-device AI workloads grow on smartphones and other mobile devices, UFS is evolving to enable the faster transfer of LLM models into DRAM. Because on-device LLM mod-

els must be loaded quickly, higher bandwidth is required; work toward UFS 5.0 targets higher link speeds by increasing the serial-bus clock rate to boost bandwidth.

Driver Overview

I proposed a UFS device driver on the freebsd-hackers mailing list in July 2024 and began analysis and design in January 2025. Fortunately, Warner Losh, now my mentor, replied and provided valuable guidance on the FreeBSD storage stack. The FreeBSD Handbook and BSD conference talks were also helpful. After about two months of analyzing CAM, SCSI, and the NVMe driver, I designed the UFS driver. Because NVMe and UFS share a similar structure, I reused many of the same ideas. I requested an early code review on May 16, 2025. After several review rounds, the work was committed on June 15, 2025, and included in FreeBSD 15.0.

Development was conducted primarily using QEMU's UFS emulation, and was later validated on real hardware: Intel Lakefield and Alder Lake platforms with UFS 2.0/3.1/4.0 devices. I also aimed to test on ARM SoCs, but suitable hardware was difficult to obtain.

The UFS driver is tightly integrated with the CAM subsystem. During initialization, it registers with CAM; thereafter, SCSI commands for configuration and reads/writes are delivered from CAM to the driver. To maintain backward compatibility with UFS 3.0 and earlier, the driver supports both the single doorbell queue (SDQ) path and the multi-circular queue (MCQ) path.

The UFS driver is tightly integrated with the CAM subsystem.

Device initialization and registration

Initialization follows the UFS layered architecture, proceeding from the bottom layer upward.

- **UFSHCI Registers:** Enable the host controller, program required registers, and enable interrupts.
- **UIC (UFS InterConnect Layer):** Issue the Link Startup command to bring up the link between the host and device and verify connectivity.
- **UTP (UFS Transport Protocol Layer):** Create the UTP command queues and enable UTP interrupts. Issue a NOP UPIU command to verify the transport path.
- **Configure Gear and Lane:** Negotiate gear and lane counts, then configure the link to operate at maximum bandwidth.
- **UAP (UFS Application Layer):** Register with CAM to begin SCSI-based initialization; CAM then scans the bus for targets and LUNs and delivers SCSI commands to the driver.

CAM (Common Access Method) is FreeBSD's storage subsystem. It is organized into three layers: the CAM Peripheral layer, the CAM Transport layer (XPT), and the CAM SIM layer. After initialization, the UFS driver creates a SIM object with cam_sim_alloc() and registers it with the XPT via xpt_bus_register(). The XPT then scans the bus for targets and LUNs to discover SCSI devices. When it finds a valid LUN, it calls cam_periph_alloc() to create and register a Direct Access (da) peripheral in the CAM Peripheral layer.

With the Direct Access (da) peripheral registered, the CAM Peripheral layer automatically constructs SCSI commands when I/O to the UFS disk is requested. The driver's ufshci_cam_action() handler, registered with the SIM, receives the CCBs that carry these commands, encapsulates them in UPIUs, enqueues them on the UTP queue, and on completion calls xpt_done() to notify the XPT.

Because CAM handles standard SCSI paths such as scanning, queuing, error handling, and retries, much of the required logic does not need to live in the UFS driver. The driver primarily forwards SCSI commands to the target device over UTP.

Queue architecture: SDQ and MCQ

One of the key design decisions was the queue architecture. UFS 4.0 introduced multi-circular queues (MCQ), which are conceptually similar to NVMe's model. For backward compatibility, single doorbell queue (SDQ) support is also required, and the driver must select between SDQ and MCQ at runtime, since UFS 3.1 and earlier support only SDQ. To address this, I defined a function-pointer operations interface (ufs_qop) that abstracts queue operations so the implementation can be chosen at runtime. (MCQ is not yet implemented and will be added soon.)

Current status and future development plans

The UFS driver is under active development and currently implements a subset of UFS 4.1 features. My goal is to achieve full feature coverage, followed by power management and MCQ. At present, supported platforms are limited to PCIe-based UFS host controllers, and I plan to add support for ARM system-bus platforms as well. I also aim to track and adopt new UFS specifications promptly as they are released.

Getting Started with the UFS driver

To test the UFS driver, you typically need hardware with UFS built in. Fortunately, QEMU allows development and testing without such hardware. This section shows how to emulate a UFS device in QEMU and exercise the driver. (UFS emulation is supported starting with QEMU 8.2.)

Prepare a FreeBSD snapshot image.

```
$ wget https://download.freebsd.org/releases/VM-IMAGES/15.0-RELEASE/amd64/Latest/
FreeBSD-15.0-RELEASE-amd64-zfs.qcow2.xz
$ xz -d FreeBSD-15.0-RELEASE-amd64-zfs.qcow2.xz
```

Create a 1 GiB file to use as the backing device for the UFS Logical Unit:

```
$ qemu-img create -f raw blk1g.bin 1G
```

Launch QEMU with an emulated UFS device:

```
$ qemu-system-x86_64 -smp 4 -m 4G \
 -drive file=FreeBSD-15.0-RELEASE-amd64-zfs.qcow2,format=qcow2 \
 -net user,hostfwd=tcp::2222-:22 -net nic -display curses \
 -device ufs -drive file=/home/jaeyoon/blk1g.bin,format=raw,if=none,id=luimg \
 -device ufs-lu,drive=luimg,lun=0
```

On amd64, the GENERIC kernel config includes the UFS driver module (see sys/amd64/conf/GENERIC):

```
# Universal Flash Storage Host Controller Interface support
device          ufshci                  # UFS host controller
```

To load the module explicitly, edit /boot/loader.conf:

```
ufshci_load="YES"
```

After reboot, verify that the UFS device is attached as ufshci0/da0 via camcontrol:

```
$ camcontrol devlist -v
scbus2 on ufshci0 bus 0:
<QEMU QEMU HARDDISK 2.5+>          at scbus2 target 0 lun 0 (pass2,da0)
<>                                 at scbus2 target -1 lun ffffffff ()
```

Basic performance checks with fio:

```
$ fio --name=seq_write  --filename="/dev/da0" --rw=write --bs=128k --iodepth=4
--size=1G --time_based --runtime=60s --direct=1 --ioengine=posixaio --group_reporting
$ fio --name=seq_read  --filename="/dev/da0" --rw=read --bs=128k --iodepth=4 --size=1G
--time_based --runtime=60s --direct=1 --ioengine=posixaio --group_reporting
$ fio --name=rand_write  --filename="/dev/da0" --rw=randwrite --bs=4k --iodepth=32
--size=1G --time_based --runtime=60s --direct=1 --ioengine=posixaio --group_reporting
$ fio --name=rand_read  --filename="/dev/da0" --rw=randread --bs=4k --iodepth=32
--size=1G --time_based --runtime=60s --direct=1 --ioengine=posixaio --group_reporting
```

QEMU is an emulator, so it is best for checking functional behavior. For performance measurements, I used my Galaxy Book S.

The Galaxy Book S has an Intel 10th-gen i5-L16G7 (1.4 GHz, 5 cores) and an internal UFS 3.1 device, which I upgraded to UFS 4.0 for the experiment (operating at HS-Gear 4 on a 3.1 host controller).

Queue Depth	Sequential Read (MiB/s)	Sequential Write (MiB/s)	Random Read (kIOPS)	Random Write (kIOPS)
1	709	554	7.1	12.1
2	1,395	556	14.8	29.4
4	1,416	559	31.6	68.2
8	1,417	554	63.5	102.3
16	1,399	555	103.7	105.5
32	1,361	556	114.2	106.6

Table 1. FreeBSD UFS Performance

Depending on queue depth, sequential write peaks at 559 MiB/s, and sequential read reaches 1,417 MiB/s, which is highly competitive for mobile devices.

Queue Depth	Sequential Read (MiB/s)	Sequential Write (MiB/s)	Random Read (kIOPS)	Random Write (kIOPS)
1	542	479	6.1	11.0
2	1,358	548	13.0	21.0
4	1,351	550	29.7	53.1
8	1,352	550	61.1	84.1
16	1,351	552	119.0	114.0
32	1,355	553	142.0	120.0

Table 2. Linux UFS Performance

Under the same conditions on Linux, performance is at a comparable level.

Conclusion

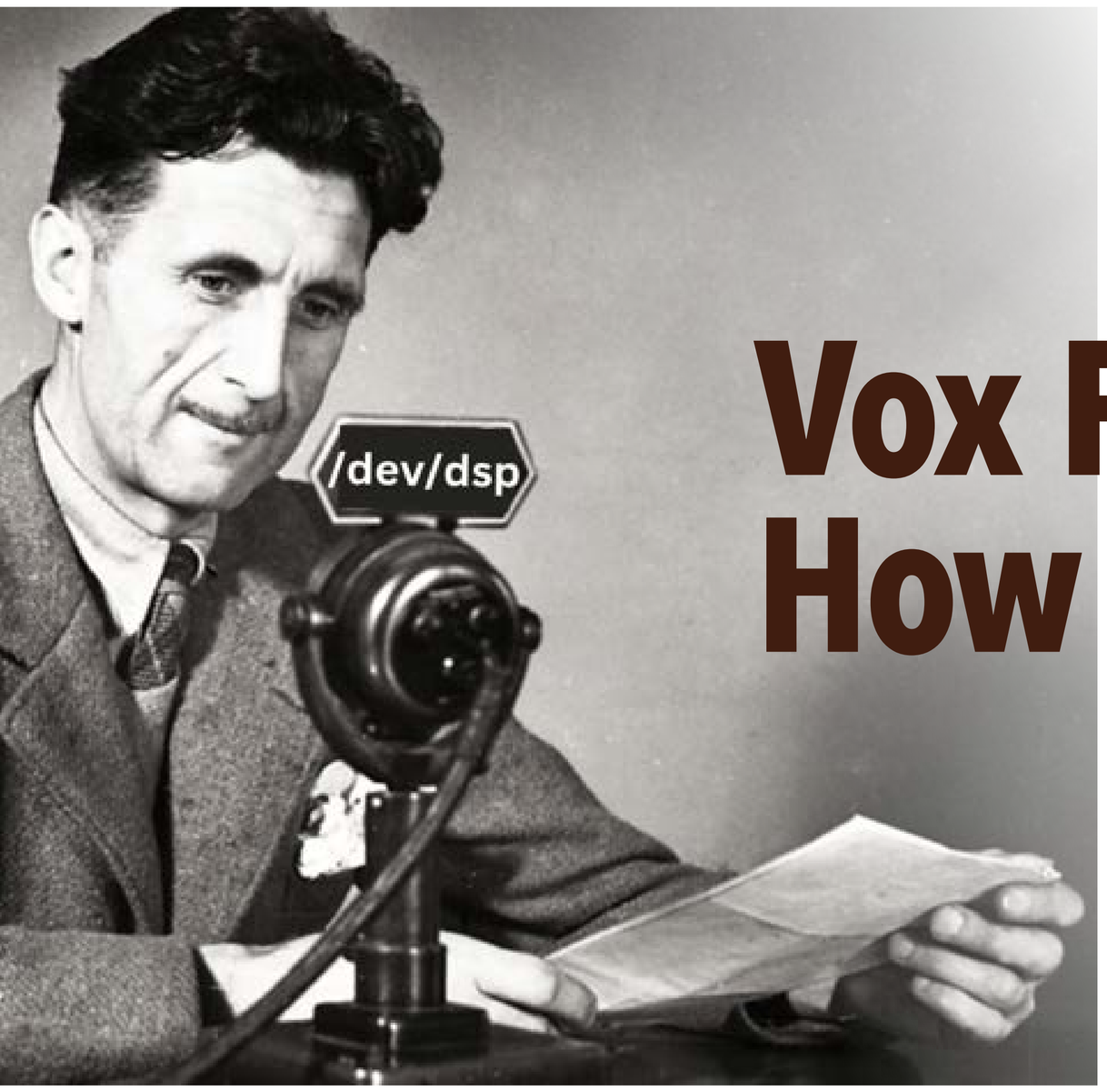
UFS is a rapidly evolving interface standard. The FreeBSD UFS driver likewise adds new features and is continually optimized to enable UFS across a variety of devices.

I hope this article encourages wider use of UFS on FreeBSD. Contributions to the UFS driver are very welcome. I'm grateful to the reviewers who helped make this possible, and I

plan to continue contributing to the community.

Development of the ufshci(4) UFS device driver was supported by Samsung Electronics.

JAEYOON CHOI is a software engineer in the Memory Division at Samsung Electronics, working to expand the open-source ecosystem for SSDs and UFS. He started using FreeBSD in 2024 and became a FreeBSD src committer in September 2025. He previously contributed to Fuchsia OS's F2FS file system and currently maintains the Fuchsia OS UFS driver. He is interested in open source for storage systems.



Vox FreeBSD: How Sound Works

BY CHRISTOS MARGIOLIS

Sound support for FreeBSD began in 1993, when Jordan K. Hubbard imported the generic Linux sound driver into FreeBSD, later known as the VoxWare sound drivers, written by Hannu Savolainen. Several new versions of the VoxWare drivers were imported (and modified) into the FreeBSD base system between 1993 and 1997. At this time, Amancio Hasty and Jim Lowe did most of the work on sound support in FreeBSD. The VoxWare drivers eventually became what we know today as OSS, the Open Sound System, maintained by Hannu and his team at 4Front Technologies.

However, things changed in 1997 as more of the foundation for the sound system we have today was laid out by Luigi Rizzo. In 1999, Cameron Grant rewrote the sound system for FreeBSD 4.0, now using the newbus interface, which supported a great deal of hardware. This superseded Luigi's driver, which was removed from the tree a month later. In the following years, many things changed in the sound area. The number of drivers for different hardware chips increased drastically (especially for PCI and USB devices), and many improvements were made to the sound infrastructure, thanks to Cameron Grant and Orion Hodson. Cameron passed away in 2005, a major loss for the FreeBSD community.

Ariff Abdullah took over the maintainership of the sound code in FreeBSD in 2005. Since then, we've seen some dramatic changes in sound support, as well as several rounds of device driver restructuring.

Ariff Abdullah took over the maintainership of the sound code in FreeBSD in 2005. Since then, we've seen some dramatic changes in sound support, as well as several rounds of device driver restructuring.

OSS

FreeBSD's sound API is called OSS, which stands for "Open Sound System". Interestingly enough, it also used to be Linux's default sound API, until it was eventually replaced by ALSA. FreeBSD, however, still uses it, and it provides a simple and clean API by just exposing

standard device files (`/dev/dsp*` corresponding to sound cards and `/dev/mixer*` to mixers), which are operated on using a few common POSIX system calls:

Syscall	Description
`open(2)`	Open device.
`close(2)`	Close device.
`read(2)`	Record audio.
`write(2)`	Play audio.
`ioctl(2)`	Query/manipulate settings (sample rate, format, volume, and much more).
`select(2)`, `poll(2)`, `kqueue(2)`	Poll for events. `kqueue(2)` is FreeBSD-only (15.0 onwards).
`mmap(2)`	Memory-mapped I/O.

The official manual can be found here: http://manuals.opensound.com/developer/

Since interaction with the sound device is done using common syscalls, it is possible to do things like this in the terminal:

1. Play white noise: `cat /dev/random > /dev/dsp`
2. Record raw PCM (Pulse Code Modulation) stream into a file: `cat /dev/dsp > foo`
3. Very crude input monitoring, which can be used to make sure your microphone is working: `cat /dev/dsp > /dev/dsp`

For more elaborate things, however, actual programs have to be written — Fortunately, the OSS API is straightforward to use. `/usr/share/examples/sound` contains several example programs and is, in fact, growing as of the writing of this article.

In addition to the `/dev/dsp*` devices, there is a `/dev/dsp` device, which is what was used in the examples above. This isn't a real device, but an alias to the currently default device. Applications that want to use whatever the default device is at any given time are strongly encouraged to access this device instead of hardcoding specific ones.

An additional nice feature is that OSS works with POSIX syscalls only is that it is trivial to use OSS in any programming language, without the need for language bindings. The only thing one would need to do to fully "port" OSS to another language would be to define some global OSS API constants and structures, which can be found in `/usr/include/sys/soundcard.h`.

sound(4)

The OSS API is implemented inside `sound(4)`. It abstracts some generic functionality into a kernel module, so that individual device drivers (e.g., `snd_hda(4)`, `snd_uaudio(4)`, etc.) do not need to duplicate code that would be the same across all of them. This generic functionality includes:

- Device creation, deletion, and access.
- Buffer management.
- Audio processing.
- Providing global and (most of) device-specific sysctls.

Device drivers have to attach to `sound(4)` and set up a communication pipeline with it, during their initialization stage. In other words, `sound(4)` is the bridge between userland

and device drivers. This is particularly convenient, because the FreeBSD kernel exposes the same **/dev/dsp*** files and sysctls (**hw.snd.*** and **dev.pcm.***) for every connected sound device, and provides uniform access and configuration to users and application programmers, while also avoiding the need for duplication at the device driver level.

sound(4) also exposes **/dev/sndstat**, which provides information about all connected sound devices and active channels, and is used internally by **virtual_oss(8)** and **sndctl(8)**.

sound(4) works with PCM audio streams, the specifics of which (sample rate, sample format, etc.) can also be manually configured by the user with **sndctl(8)**, if needed. This means that applications working with audio represented in other formats, such as WAV, Opus, MP3, etc., need to convert the stream to PCM during playback, and from PCM during recording.

High-level implementation overview

Device access

As was mentioned earlier and shown in the example program, applications access the sound subsystem through **/dev/dsp***, using the **open(2)** syscall. The following flags can be specified:

open(2) flag	Description
O_RDONLY	Recording.
O_WRONLY	Playback.
O_RDWR	Recording and playback.

The **O_NONBLOCK** and **O_EXCL** flags can be additionally specified in the **open(2)** call, for non-blocking I/O and exclusive access to the device respectively. For **sound(4)** to know which channels belong to which file descriptor, and to route audio and information properly, it uses **DEVFS_CDEVPRIV(9)**.

As alluded to earlier, there are also **/dev/mixer*** devices. This is a legacy interface, mainly used for volume setting and recording source selection, by applications such as **mixer(8)**. **/dev/mixer*** devices have a 1:1 relationship with **/dev/dsp*** ones, so for instance, **/dev/mixer0** is the mixer device corresponding to **/dev/dsp0**. This interface also provides some functionality for physical mixers. As of OSS version 4.0, the mixer API has been rewritten, but is not currently fully implemented on FreeBSD.

Channels

Channels hold some important information about their state (e.g., configuration, PID, and name of process consuming it, diagnostic values, etc.), as well as the most important component; the buffer(s). In other words, the actual audio stream. Channels also have their own volume, in addition to the device-wide master volume. This is especially useful because applications get their own volume knobs and usually do not need to touch the master one.

At this point, it is essential to note that there are two types of channels in **sound(4)**; primary/"hardware" and virtual.

The device drivers allocate primary channels, and there is *usually* one for playback and one for recording, depending on what is supported by the hardware. However, some drivers might make the number of primary channels equal to the number of physical playback and recording ports provided by the hardware. Each primary channel comes with a pair of

buffers, a software-facing one and a hardware-facing one. The software-facing buffer is responsible for exchanging audio data with userland, while the hardware-facing one is for exchanging data with the hardware. Whenever the device driver is ready to read from or write data to the hardware, it interrupts **sound(4)**, and data is copied from one buffer to the other. During playback, since we are writing data *to* the hardware, we copy the software-facing buffer to the hardware one, so that the driver can feed that data to the hardware. The reverse is true for recording.

Virtual channels, commonly referred to as VCHANs, are treated as children of primary channels, but, unlike primary channels, they do not have any connection to the hardware, so only their software-facing buffer is used. The reason for virtual channels existing in the first place is that we want an indefinite number of applications to be able to access the device simultaneously. Without virtual channels, the number of processes that can access the device simultaneously is equal to the number of primary channels, since each channel has only one software-facing buffer, which means that all processes would have to share the same buffer, which is not really ideal for audio, so each buffer has to be dedicated to one process.

The reason for virtual channels existing in the first place is that we want an indefinite number of applications to be able to access the device simultaneously.

Earlier, we explained how audio streams are exchanged between userland and hardware using primary channels. When virtual channels are enabled (as is the case by default), **sound(4)**, instead of simply copying the primary channel's software-facing buffer to the hardware-facing one (during playback), and vice-versa (during recording), it first has to mix all the audio streams of the primary channel's children virtual channels, and then supply the final mixed stream to the primary channel's hardware-facing buffer. As you can imagine, this additional layer introduces a slight overhead, which is irrelevant for most use cases, but might not be ideal in some low-latency music production workflows. For those cases, virtual channels can simply be disabled.

To view channel states, you can use **sndctl(8)**:

```
$ sndctl -v
```

Processing chain

An interesting feature of **sound(4)** is its processing chain. This includes:

- Mixing. This is actually exactly what was explained in the previous section, about how virtual channel streams are (de-)mixed.
- Volume control.
- Channel matrixing. **sound(4)** is capable of doing any-to-any channel matrixing, for example, mono to stereo, or stereo to 5.1 surround. This is done by converting streams from one interleaved PCM format to another.
- Basic parametric equalization.
- Format conversions.

- Resampling. There are three different resampling types, namely:
  - Linear.
  - Zero-order-hold (ZOH).
  - Sine Cardinal (SINC).

Each channel gets its own processing chain, and it includes only the necessary components. For instance, if the channel is configured to have a sample rate of 44100Hz, but the application feeds it audio sampled at 48000Hz, then **sound(4)** will need to include resampling in the channel's processing chain. Similarly, if the stream has the same sample rate as the channel, then that component will not be needed. The same applies to the other components.

A helpful way to visualize the processing chain is to print it using **sndctl(8)**. The following command will print the chain of each active channel:

```
$ sndctl feederchain
```

In the next section, we will discuss how and why, in some specialized cases, you might want to bypass the processing chain entirely.

Memory-mapped I/O and bit-perfect audio

Two of **sound(4)**'s liked features by low-latency application developers and audio enthusiasts, are that it provides bit-perfect mode support, as well as memory-mapped I/O.

Bit-perfect mode means that the audio stream skips all of **sound(4)**'s processing chain, and is fed more or less directly to the sound card. Applications have the added responsibility of making sure the stream's configuration (sample rate, format, channel matrix) matches that of the sound card. For instance, if the application wants to play audio sampled at 48000Hz, but the sound card does not support that sample rate, then it needs to take care of resampling the stream. As a result, this feature is disabled by default and is enabled only by applications that implement their own processing, and/or users who are sure their sound card will work properly in bit-perfect mode.

Memory-mapped I/O is similar to bit-perfect, in that the audio stream skips all processing done by **sound(4)**; in fact, bit-perfect has to be enabled in order to do memory-mapped I/O. However, the major difference between bit-perfect and memory-mapped I/O, is that the latter puts all of the audio buffer handling responsibilities entirely on the application, which means that it has to take care of not only the same things that an application using bit-perfect would, but to also make sure the buffer is synchronized correctly and that read/writes happen at the right time, with some help from **sound(4)**. If done right, and in the right environment, this can yield performance improvements, but is quite tricky and tedious to implement correctly, and so is mostly discouraged, unless the programmer really knows what they are doing.

Device drivers

Just like **sound(4)** is the bridge between userland and the device drivers, the device drivers are the bridge between **sound(4)** and the actual hardware. Apart from the fact that all sound drivers attach to **sound(4)** and communicate with it, the rest of their functionality depends on the driver itself. In a future article, we could present how to write a sound driver from scratch.

FreeBSD ships with support for the following sound cards:

Driver	Soundcards	Enabled by default
`snd_ai2s(4)`	Apple I2S	powerpc
`snd_als4000(4)`	Avance Logic ALS4000	
`snd_atiixp(4)`	ATI IXP	
`snd_cmi(4)`	CMedia CMI8338/CMI8738	amd64, i386
`snd_cs4281(4)`	Crystal Semiconductor CS4281	
`snd_csa(4)`	Crystal Semiconductor CS461x /462x/4280	amd64, i386
`snd_davbus(4)`	Apple Davbus	powerpc
`snd_emu10k1(4)`	SoundBlaster Live! and Audigy	
`snd_emu10kx(4)`	Creative SoundBlaster Live! and Audigy	amd64, i386
`snd_envy24(4)`	VIA Envy24 and compatible	
`snd_envy24ht(4)`	VIA Envy24HT and compatible	
`snd_es137x(4)`	Ensoniq AudioPCI ES137x	amd64, i386
`snd_fm801(4)`	Forte Media FM801	
`snd_hda(4)`	Intel High Definition Audio	amd64, i386
`snd_hdsp(4)`	RME HDSP	
`snd_hdspe(4)`	RME HDSPe	
`snd_ich(4)`	Intel ICH AC'97 and compatible	amd64, i386
`snd_maestro3(4)`	ESS Maestro3/Allegro-1	
`snd_neomagic(4)`	NeoMagic 256AV/ZX	
`snd_solo(4)`	ESS Solo-1/1E	
`snd_spicds(4)`	I2S SPI	
`snd_t4dwave(4)`	Trident 4DWave	
`snd_uaudio(4)`	USB audio and MIDI	auto-loaded on device plug
`snd_via8233(4)`	VIA Technologies VT8233	amd64, i386
`snd_via82c686(4)`	VIA Technologies 82C686A	
`snd_vibes(4)`	S3 SonicVibes	

There is also support for the following ARM chips:

- Allwinner A10/A20 and H3.
- Broadcom BCM2835.
- Freescale Vybrid.
- Freescale i.MX6.

If you own a sound card whose driver is not enabled by default on your machine's architecture, or you are using a custom kernel configuration without sound compiled in, and are unsure which driver your sound card uses, you can run the following command:

```
# kldload snd_driver
```

`snd_driver` is a meta-driver that loads all available drivers. Once you figure out which driver attaches to your sound card, you can load that one only.

Recent improvements

The sound subsystem has (and still is) undergone many improvements in the last two years, including a number of bug and crash fixes, the introduction of a growing Kyua test suite and a testing driver (`snd_dummy(4)`), as well as multiple cleanups and refactors.

A few important user-facing improvements include:

- Hot-unplugging is now possible. Users of USB sound cards on older versions of FreeBSD might remember that hot-unplugging the sound card usually resulted in the USB bus hanging, until the application using the now-detached device was manually killed (PR 194727).
- Floating-point audio support. This is a bit misleading, though, because we do not really, at least currently, support floating-point audio on the device driver level, but rather, we allow userland applications to use OSS with floating-point audio. This already fixes quite a few ports, such as Wine, that needed floating-point audio support from OSS.
- `sound(4)` now only exposes a single `/dev/dsp*` file for each device and does all the audio stream routing internally, using `DEVFS_CDEVPRIV(9)`, as opposed to exposing a `/dev/dsp*` file for each allocated audio stream. The current approach is cleaner both in implementation and in what is exposed to userland.
- Better out-of-the-box support for High Definition Audio (`snd_hda(4)`) sound cards. These cards are a constant pain for both developers and users, because they tend to come with non-standard configurations, meaning that we have to compensate for that by adding manual patches inside the driver or `/boot/device.hints`. A commonly reported issue is that sound is not automatically redirected to the headphones once they are plugged in, and vice versa. Several patches have recently been written for various sound cards that experience this issue, especially Framework laptops. It is very likely that you also have fallen victim to that issue. With that being said, since FreeBSD 15.0, there is a `devd(8)` configuration, `/etc/devd/snd.conf`, which attempts to automate this issue. The basic idea of the implementation is that whenever `snd_hda(4)` detects that a jack has been (un-)plugged, it issues a `devd(8)` notification, and `/etc/devd/snd.conf` will make sure to redirect sound to the appropriate device using `virtual_oss(8)`. This feature is still experimental, so there should be more refining as more people provide feedback.
- kqueue(2) support for `sound(4)`.

Userland utilities

You can find examples and more information for each of the following utilities in their respective manual pages.

sndctl(8)

sndctl(8) lists and manipulates sound card settings, such as the sample rate, sample format, bit-perfect and realtime mode settings, among others. It aims to be a replacement for **/dev/sndstat** (in fact, it uses it internally) and some of **sound(4)**'s sysctls, at least for most use cases:

```
$ sndctl
pcm3: <Realtek ALC295 (Analog 2.0+HP/2.0)> on hdaa1 (play/rec)
    name                 = pcm3
    desc                 = Realtek ALC295 (Analog 2.0+HP/2.0)
    status               = on hdaa1
    devnode              = dsp3
    from_user            = 0
    unit                 = 3
    caps                 = INPUT,MMAP,OUTPUT,REALTIME,TRIGGER
    bitperfect           = 0
    autoconv             = 1
    realtime             = 0
    play.format          = s16le:2.0
    play.rate            = 48000
    play.vchans          = 1
    play.min_rate        = 1
    play.max_rate        = 2016000
    play.min_chans       = 2
    play.max_chans       = 2
    play.formats         = s16le,s32le
    rec.rate             = 48000
    rec.format           = s16le:2.0
    rec.vchans           = 1
    rec.min_rate         = 1
    rec.max_rate         = 2016000
    rec.min_chans        = 2
    rec.max_chans        = 2
    rec.formats          = s16le,s32le
```

mixer(8)

mixer(8) deals with volume control, (un-)muting, recording source(s) selection, and default device setting. It was completely rewritten on FreeBSD 14.0, and comes with an improved interface and functionality:

```
$ mixer
pcm3:mixer: <Realtek ALC295 (Analog 2.0+HP/2.0)> on hdaa1 (play/rec) (default)
    vol          = 0.75:0.75     pbk
    pcm          = 1.00:1.00     pbk
    rec          = 0.37:0.37     pbk
    ogain        = 1.00:1.00     pbk
    monitor      = 0.67:0.67     rec src
```

virtual_oss(8)

virtual_oss(8) is a powerful sound server for OSS written by the late Hans Petter Selasky . It was part of ports (**audio/virtual_oss**) for years, but has been part of the base

system since FreeBSD 15.0. It is again in active development, and there are already plenty of significant improvements being worked on and planned for the future.

As is mentioned in the 15.0 release notes, pre-FreeBSD 15.0 users of **virtual_oss(8)** can simply uninstall the **audio/virtual_oss** port and use the base system version. The only thing to keep in mind is that some functionality, which depends on third-party libraries, has been moved to separate ports, namely:

- **sndio** backend support: **audio/virtual_oss_sndio**
- bluetooth backend support: **audio/virtual_oss_bluetooth**
- virtual_equalizer(8): audio/virtual_oss_equalizer

mididump(1)

mididump(1) is a simple utility that prints MIDI events for a given device in real time. This is useful for making sure a MIDI device works properly and that keys work and are mapped correctly.

```
$ mididump /dev/umidi0.0
Note on                 channel=1, note=53 (F3, 174.61Hz), velocity=109
Note off                channel=1, note=53 (F3, 174.61Hz), velocity=127
Note on                 channel=1, note=55 (G3, 196.00Hz), velocity=100
Note off                channel=1, note=55 (G3, 196.00Hz), velocity=127
Pitch bend              channel=1, change=1
```

beep(1)

beep(1), as the name suggests, plays a beep sound. This is an easy way to verify sound works.

More

Apart from the utilities mentioned, there are a few more things provided by the sound subsystem:

What	Description	Documentation
mixer(3)	A C library for interacting with the OSS mixer.	**man 3 mixer**
sndstat(4)	An **nv(9)** interface for listing device information, as well as registering userland sound devices. Used internally by **sndctl(8)** and **virtual_oss(8)**.	**man 4 sndstat**
hw.snd.*	Global **sysctl(8)** variables.	**man 4 sound**
dev.pcm.*	Device-specific **sysctl(8)** variables.	**man 4 sound**
Driver-specific **sysctl(8)** variables		Refer to the respective driver's manual page.

FreeBSD for music production?!

You might be thinking this is a joke, but it is, in fact, a topic that has been coming up more and more in recent years, and we have already seen a few related talks in recent conferences, more specifically:

- Goran Mekić, FOSDEM 2019
- Goran Mekić, EuroBSDCon 2022
- Charlie Li, BSDCan 2024
- Christos Margiolis, FreeBSD DevSummit 09/2024
- Christos Margiolis, BSDCan 2025
- Charlie Li, EuroBSDCon 2025

FreeBSD is, without a doubt, not the operating system a musician or producer would typically think about when it comes to music production, however, this is partially the case because of a lack of "marketing", for lack of a better word. In reality, FreeBSD offers a solid, fast, and highly configurable sound subsystem, it has a consistently rapidly growing collection of open source Digital Audio Workstations, LV2 plugins, and other types of production/music software, and it can work with any non-native sound subsystem (ALSA, sndio, JACK, Pulseaudio, Pipewire, etc.), in case OSS is not desirable.

I genuinely think that if we continue this trend of consistently maintaining and developing the sound subsystem, porting and developing more software, as well as showcasing in practice why FreeBSD can be an alternative for audio and music production, both in conferences and online, we could, one day, see FreeBSD gaining significant popularity among audiophiles and musicians.

Reporting and resolving bugs

All software might contain bugs from time to time, and the sound subsystem is no exception. Providing sufficient information is always necessary, and opening a bug report or sending an email with just a "sound does not work on my machine" is not really helpful. Attaching the output of the following commands should be enough in most cases:

1. `uname -a`
2. `sndctl -v`
3. `mixer -a`
4. `sysctl hw.snd dev.pcm`, as well as the driver-specific sysctls, if any.
5. `dmesg`, after setting `hw.snd.verbose=4` and reproducing the bug.
6. Logs, if any, from the application with which the bug is reproduced.

Conclusion

Hopefully, this article has helped with presenting the general structure of the sound subsystem as a whole, at its current state. It would be great to see even more people interested in FreeBSD sound in the future! The `freebsd-multimedia@FreeBSD.org` mailing list is where most of the discourse happens, so make sure to keep an eye on it.

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

Support FreeBSD®

Donate to the Foundation!

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

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

Your investment will help:

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

Making a donation is quick and easy.

freebsdfoundation.org/donate

FreeBSD™
FOUNDATION

Credentials Transitions with `mdo(1)` and `mac_do(4)`

BY OLIVIER CERTNER

In this article, we explore how the **`mdo(1)`** program can be used to easily and quickly launch a new process with different credentials and how system administrators can enable credentials transitions initiated by unprivileged users by leveraging the **`mac_do(4)`** kernel module, obviating the need to install third-party programs such as **`sudo(8)`** or **`doas(1)`** in simple role-based scenarios.

The traditional UNIX approach to access control essentially relies on the following concepts and components:

- Users and groups. Groups are meant to ease administration by treating all users of a group uniformly in some ways.
- Processes, as subjects acting on behalf of some user and groups, which are referred to as their credentials.
- File ownership (one user, one group) and permissions, which separately guard accesses by the owner, by members of the file's group, and by other users.
- The special **`root`** user[1], which has all privileges and in particular is not subject to access control.
- Set-user-ID/set-group-ID executables, which, when launched, have their process respectively endorse the executable's owner as the user and the executable's group as the "primary" group[2].

One of the primary duties of a system administrator is to give their users appropriate access to various resources of the systems. This, for the most part, translates into defining users and groups and ensuring that files have the proper permissions with respect to the expected security policy.

The UNIX access control model has the flexibility that users need not represent real, human individuals, but may as well represent roles that can be assumed by multiple real users who require access to specific resources and information. Such a role-based approach relying on UNIX users instead of just groups is, in fact, necessary in all but the simplest file-sharing scenarios. It makes temporarily adopting another set of credentials, established from a target user, an important feature of the system, which the **`su(1)`** program traditionally fulfills.

One of the primary duties of a system administrator is to give their users appropriate access to various resources of the systems.

However, **`su(1)`** requires authenticating to a new user before switching to its credentials, typically by asking for the user's password[3], which is not convenient either for humans who have been assigned roles and are already authenticated or for automated scenarios. It nec-

essarily spawns the target user's shell, which precludes using it with users who have no valid login shells, whereas this is typically desired for role users that nobody should be able to log in as directly. It also makes launching a specific program with arguments more cumbersome than it should be[4].

In order to overcome these limitations, system administrators commonly install other programs designed to run commands on behalf of other users such as **sudo(8)** or **doas(1)**. However, programs like **sudo(8)** have a non-negligible attack surface, in part due to their number of infrequently used features, and, in particular, modularity, which can be dangerous from a security standpoint. More generally, having programs installed with the executable file's owner being **root** and having the set-user-ID mode bit set is a security concern, as compromising them can mean gaining full administration rights through code execution as the **root** user. But traditional UNIX did not provide any other way to change credentials, which is why programs like **su(1)** and **login(1)** are installed this way.

As an alternative to executables with the set-user-ID mode bit set, more colloquially called "setuid executables", we provide the **mac_do(4)** kernel module, built on top of FreeBSD's MAC framework[5]. Its purpose is to allow only certain credentials transitions from unprivileged processes, thus not requiring the corresponding executable image to be installed "setuid".

As an alternative to "setuid executables" in role-based settings, we provide the **mac_do(4)** kernel module and its **mdo(1)** companion program.

mdo(1) is the companion program to **mac_do(4)** and actually requests the desired credentials transitions to the kernel. **mdo(1)** can be used standalone by the **root** user who has all privileges. Otherwise, its requests are vetted by the **mac_do(4)** kernel module according to the administrator's configuration.

In this article, we first describe how to use **mdo(1)** to launch commands under new credentials, walking through a series of examples. Then, we explain how to configure **mac_do(4)** to enable specific credentials transition in support of role-based schemes, on the host system as well as in jails, also offering some insights on the current design. Finally, we ask for user feedback on what should come in the relative short term and possible longer term future plans.

Using mdo(1)

mdo(1) is designed to run any command with an arbitrary set of credentials. If you have not yet configured **mac_do(4)**, which is covered in the next section, you can still run all the examples below as **root**. For most examples, FreeBSD 15.0 is required as FreeBSD 14.3's **mdo(1)** only supports options **-u** and **-i**[6].

For safety reasons, the target process credentials have to be fully specified, either explicitly by listing all users and groups and their requested values, or indirectly by establishing a baseline that provides a default value for each of them and can then be amended by additional options.

In a role-based setting, the most common use case is arguably to endorse the credentials of some user as if he has just logged in. So, **mdo(1)** supports it in the simplest form

possible, with the only needed option being **-u** (for "user") to establish a baseline from the named user, as in:

```
$ mdo -u www /usr/local/bin/occ
```

To **sudo(8)** and **doas(1)** users, this command-line should strike you as extremely similar to what you use with those tools: Basically, **mdo** replaces **sudo** or **doas**, and the rest is identical.

Obviously, this cannot work if passing some user numerical ID as opposed to a user name, as the full login credentials are determined by the password and group databases, which are indexed by names[7]. Using **-u** with a user numerical ID only specifies the user (actually, the real, effective and saved user IDs), and **mdo(1)** then needs to be told about all target groups. That has to be done either explicitly as we will see below, or through **-i** which means to use the current groups as a baseline, else **mdo(1)** will simply error.

Keeping the current groups, such as in:

```
$ mdo -u 10002 -i
```

can be useful, e.g., if you need to temporarily process alien files whose owner is not in your password database.

-i also works with **-u** with a user name, and it means the same. E.g., if **foo** is some user in your password database that has ID 10002, then this command is entirely equivalent to the previous example:

```
$ mdo -u foo -i
```

Suppose now you want to explicitly specify groups, either because you are using a numerical user ID as we saw above, or because you want to override the groups associated with some user. You can use:

- **-g**: To set/override the primary groups[8].
- **-G**: To set/override the full set of supplementary groups. The comma-separated list you provide here is considered complete, i.e., it should contain all supplementary groups. Keep in mind that, starting with FreeBSD 15, a user logging in has their initial group, as specified in the password database, also in its processes' supplementary groups set.
- **-s**: To amend the supplementary groups set. The argument for this option consists of a list of comma-separated directives. You can ensure a group is part of the supplementary groups with a **+** directive, or ensure it is not with a **-** directive, or reset the list with a **@**, making **-s** work like **-G** but with a different syntax[9].

Some examples:

```
$ mdo -u unprivileged_user -g wheel -G wheel,staff,operator
```

starts a shell as user **unprivileged_user**, but ignores the groups specified in the password and group databases, replacing them with the passed ones. **-g** and **-G** are useful for testing which rights would be given to a user with a particular set of groups.

However, if the point is only to log in as some user with some additional groups, e.g., **wheel** and **operator**, in an "augmented" role scenario, **-s** is the option to use:

```
$ mdo -u unprivileged_user -s +wheel
```

or, conversely, if membership of some group must be temporarily revoked, e.g., when this group is used as a tag to deny access through **ugidfw(8)** (and **mac_bsdextended(4)**) for access control:

```
$ mdo -u unprivileged_user -s -tag_group
```

If the user should not change, there is no need to specify it explicitly with **-u**. Instead, you can just use **-k** (for "keep"), meaning to start with all current users and groups as the baseline. **-k** is exclusive with **-u** and implies **-i** (start from current groups).

Note that, in all cases, it is possible to override the credentials' groups using the explicit options we have seen (**-g**, **-G**, and **-s**).

Finally, the real, effective and saved variants of users and primary groups can be separately overridden if needed, using **--ruid**, **--euid** and **--svuid**, and **--rgid**, **--egid** and **--svgid** respectively. When all three variants are specified, there is no need to respectively use **-u** or **-g**, although specifying **-u** with a name is still useful to get its associated groups.

As you can see, in addition to its simplicity for the most common use cases, **mdo(1)** thus has the advantage over **su(1)**, **sudo(8)** or **doas(1)** that it allows to control every aspect of the target credentials, making it the tool of choice to test or temporarily use arbitrary credentials in advance of a modification of the password and group databases, or for role-based settings where endorsing a role comes from being part of additional groups rather than switching users.

mdo(1) is only concerned with changing credentials. Consequently, its code is relatively simple, and a special effort was made to make it as clear and as minimal as possible while being "obviously" correct. This makes for a program that is easy to audit and results in a very small binary, weighing a little more than 7kB on my **stable/14** machines. Compare this to **doas(1)** which weighs a little more than 27kB, and **sudo(8)** at 229kB completed by **sudoers(5)**, its default security policy plugin, at 628kB, both programs being installed as "setuid executables" in contrast to **mdo(1)**.

Currently, as it is geared to role-based scenarios, **mdo(1)** does not ask for any password or other form of authentication when requesting new credentials, instead relying solely on the requester's credentials for this purpose. As one of the possible future directions, listed in the conclusion of this article, we may add support for asking the current logged-in user's password. Additional functionalities related to switching to another user (such as login classes, login name, scheduling priorities, etc.) may also be considered depending on user feedback.

Configuring mac_do(4)

For a non-root user to be able to leverage **mdo(1)**, configuring **mac_do(4)** is compulsory since **mdo(1)** is by-design not installed "setuid".

mac_do(4) is not compiled in the kernel by default, but can easily be loaded as a module:

```
# kldload mac_do
```

You can then access its parameters below the **security.mac.do sysctl(8)** node. Currently (FreeBSD 14.3 and 15.0), the following are available:

- **enabled**: Whether the module is enabled (defaults to true). This is a global toggle. It is possible to deactivate **mac_do(4)** selectively on the host system or in any jail via rules (next knob) or jail parameters (see corresponding subsection below).
- **rules**: A list of rules indicating which credentials transitions are allowed. We are going to study several examples in the next subsection. **rules** has an empty value by default, meaning that **mac_do(4)** will not allow any credentials changes by itself.
- **print_parse_error**: Whether to print a parse error on the console and system log when setting rules fails.

Let's start by illustrating rules, and we will then get to how to configure jails.

Rules

Related to some examples we gave above for **mdo**, let's authorize user **unprivileged_user** with UID 10001 to endorse the **www** user (UID 80) representing a webmaster role:

```
# sysctl security.mac.do.rules='uid=10001>uid=80,gid=80,+gid=80'
```

In this example, there is only a single rule. The **>** token separates both parts of a rule, the left part being the "from" one, also called "match", and the right part being the "to" one, also called "target". **:** has been the historical separator token and still works, but we felt that **>** makes for more easily readable rules, especially to UNIX-trained eyes that can easily interpret **:** as a list separator between similar elements. Because **>** is a shell special character, you need to quote it somehow. For simplicity, we advise to always quote the value passed to **sysctl(8)**. Any amount of spaces can be used between tokens as another slight help to human users, and this feature also requires shell quoting.

The "from" part (**uid=10001** in the above rule) is pretty straightforward and indicates to match processes whose user ID[10] is 10001, thus matching **unprivileged_user** (and possibly other users with the same user ID). Note that only numerical IDs are allowed, not user names. The kernel indeed does not know about user names, which are irrelevant credentials-wise.

The "to" part (**uid=80,gid=80,+gid=80**) is a bit more involved. It contains three clauses separated by **,**. The **uid=80** and **gid=80** ones should be pretty straightforward: They allow switching to **www** in terms of user and initial ("primary") group ID. The last clause, **+gid=80**, is about supplementary groups, and says that 80 as a supplementary group ID is allowed but not mandatory. In general, **gid** preceded by a flag, here **+**, applies to supplementary groups. Other possible flags are **!** and **-**, and will be illustrated below.

Such a rule allows, e.g., the example command we saw in the previous section to be executed by **unprivileged_user**:

```
$ mdo -u www /usr/local/bin/occ
```

Note that the **uid=10001>uid=80,gid=80,+gid=80** rule is quite stringent, and for example would not allow **mdo -u www** to succeed if, e.g., user **www** was also a member of another group than **www**, as **mdo -u www** would try to install the supplementary groups mandated by the password[11] and group databases, and that other group does not appear in the rule.

It also forbids, e.g., **mdo -u www -i**, meaning to switch to user **www** but to keep the current groups, presumably those associated to **unprivileged_user** if they were not changed in the meantime. If an administrator wants this to work, it needs to relax the checks on groups. Assuming **unprivileged_user** is only a member of a group with the same name and GID 10001, they could use:

```
# sysctl security.mac.do.rules='uid=10001>uid=80,gid=80,gid=10001,+gid=80,+gid=10001'
```

As you can probably infer from this example, specifying multiple target clauses with **gid** and **+gid** means that any of the specified groups can be present in the target credentials[12].

In addition to the two last **mdo(1)** use cases, this last rule also allows **unprivileged_user** to become **www** while endorsing both groups 80 and 10001 at the same time[29]. If this is not desired at all, then the following setting could be used instead:

```
# sysctl security.mac.do.rules='uid=10001>uid=80,gid=80,+gid=80;uid=10001>uid=80,gid=1
0001,+gid=10001'
```

This time, there are two rules separated by `;`. When there are multiple rules, it is enough for one of them to validate the transition for it to be possible[13]. This setting still allows for mdo `-u www` and `mdo -u www -i` to work while ruling out something like `mdo -u www -i -s +www` or `mdo -u www -g 10001`.

If for some reason `mdo -u www -i` should work also when the current groups do not reflect what the databases say for user `unprivileged_user`, you can alternatively use:

```
# sysctl security.mac.do.rules='uid=10001>uid=80,gid=80,+gid=80;uid=10001>uid=80'
```

The second rule above, `uid=10001>uid=80`, allows a change of user ID without changing the current groups, so is exactly adapted to the use of `-i` with `mdo(1)` when any set of current groups can be kept. That second rule is in fact a shortcut for `uid=10001>uid=80,gid=.,!gid=.`, where `.` stands for the current primary groups[2] in the case of `gid`, and for the current supplementary groups in the case of `!gid`, and more broadly in the case of `gid` preceded by another flag. Note that this default part, `gid=.,!gid=.`, is implied only when no target clause has `gid` as its type (with or without flags). In particular, the following rule: `uid=10001>uid=80,gid=.` would prevent any switch that does not drop all supplementary groups, as no `gid` clause with flags appear.

An additional `gid` flag, `-`, can be used to indicate that a group shall not be part of the final supplementary groups. You may at first find this strange, as allowed groups have to be explicit in rules, barring the default explained in the previous paragraph. This is actually useful in conjunction with `.` used with `+gid` or `!gid`, in order to rule out some of the current groups. For example, if you want to allow `unprivileged_user` to switch to user `www` but retaining its current groups while ensuring that `wheel` does not appear in the final supplementary groups, instead of the above `uid=10001>uid=80`, you could use `uid=10001>uid=80,gid=.,+gid=.,-gid=0`[14].

Finally, in place of user or group IDs in rules, you can use `*` or `any` to mean any possible ID. For example, if you do want to allow members of `wheel` to become `root`, you could use a rule like `gid=0>uid=0,gid=*,+gid=*`, basically saying that any set of target groups will do. Going even further, if you do not want to impose switching to `root` before becoming another user, you could as well use `gid=0>uid=*,gid=*,+gid=*`, which can be abbreviated to `gid=0>any`. Let us remind you that, currently, `mdo(1)` is geared to role-based schemes and consequently, as in any other case, will not ask for a password to be entered to switch to another user, even if the latter is `root`.

When there are multiple rules, it is enough for one of them to validate the transition for it to be possible.

We have just demonstrated a large practical assortment of possibilities offered by `mac_do(4)`'s rules, which as you can see are very flexible and able to express precisely the target credentials that are to be allowed[15]. We have tried hard to keep the syntax as easy as possible to understand with the constraints of an essentially single-line `sysctl(8)` value, imply-

ing terseness, sufficient expressive power and the kernel dealing only with numerical IDs and not accessing the password and group databases. Even if you do not immediately grasp what a particular setting of **`security.mac.do.rules`** means, it should not take long before you do, so do not get overwhelmed by the examples and take some time to study them as necessary.

An exhaustive and more formal specification of rules can be found in the **`mac_do(4)`** manual page.

Jails

Jails in FreeBSD form a hierarchy[16], whose top is the host system[17]. Each individual jail has parameters, some of which can only be set at jail creation, and others also while the jail runs, from outside the jail.

`mac_do(4)` supports per-jail configuration thanks to the following parameters:

- **`mac.do`**: The per-jail module's mode.
- **`mac.do.rules`**: The per-jail rules that apply.

Parameter **`mac.do.rules`** contains the applicable rules, with exactly the same format as the **`security.mac.do.rules sysctl(8)`** knob we saw in the previous section.

It is usually desirable to control security parameters from outside a jail, and that is actually the only possibility at jail creation. However, it is also useful to have jails behave as closely as possible to the host system. Since **`mac_do(4)`** is a tool for an administrator to authorize credentials transitions, an administrator in a jail should also be able to use it.

For this reason, the **`security.mac.do.rules sysctl(8)`** knob was made jail-aware, i.e., it reflects the current jail's setting and can be set from the jail itself. **`security.mac.do.rules`** inside a jail and the corresponding jail's **`mac.do.rules`** parameter are in fact the same variable, so their values are always the same. An outside modification of **`mac.do.rules`** is immediately in force inside the jail, and conversely reading the jail parameter reveals any inner modification to **`security.mac.do.rules`**.

Parameter **`mac.do`** indicates how **`mac_do(4)`** works in a jail. As typical for the master parameter of a module supporting jails, it accepts or reports the following values:

- **`new`**: Jail's configuration is independent from that of the parent jail.
- **`inherit`**: Jail's configuration is inherited from the parent jail.
- **`disable`**: **`mac_do(4)`** is disabled in the jail.

For obvious security reasons, the default value is **`disable`**, except if **`mac.do.rules`** is explicitly set.

You may wonder what the exact interactions of this parameter with **`mac.do.rules`** are, as both parameters appear to be somewhat redundant. As said in this section's introduction, setting rules to an empty string causes **`mac_do(4)`** to ignore credentials change requests, and since rules are per-jail, this also works as a per-jail toggle to disable **`mac_do(4)`**, similarly to the **`disable`** value. Conversely, setting **`mac.do.rules`** from outside the jail, or **`security.mac.do.rules`** inside it, always has the effect of establishing per-jail settings, which conceptually corresponds to **`new`**.

We introduced[18] the **`mac.do jail`** parameter for two reasons. First, most kernel modules supporting jails provide a single knob to enable or disable its functionality inside a jail, and we found it good to have one, both for system consistency but also to provide a perhaps more natural way of disabling **`mac_do(4)`** than setting the rules to an empty string. Second, it

gives us the opportunity to introduce a new inheritance mode, through the **`inherit`** value, which can be very useful to administrators who want a set of jails to behave the same.

Before examining what inheritance exactly means, let's first see how **`mac.do`** and **`mac.do.rules`** stay consistent. Internally, each jail holds a kind of flag indicating whether it inherits from its parent and, if it does not, a copy of the rules setting (**`mac.do.rules`**) and an internal representation for them, avoiding any information redundancy[19]. We do not actually store any value that directly corresponds to the **`mac.do`** parameter. Rather, the latter gets synthesized from the available data when it is read. After this description, you probably have an idea of how it goes: If the inheritance flag is set, then reading **`mac.do`** returns **`inherit`**, else, if no rules were specified (empty string), **`disable`**, else **`new`**. When setting **`mac.do`** explicitly, **`mac_do(4)`** checks that its value is consistent with that of **`mac.do.rules`**. If **`mac.do`** is set to **`new`**, **`mac.do.rules`** must be specified. For the other cases, we apply the robustness principle[20], tolerating the presence of **`mac.do.rules`** with an empty string in jail parameters, even if strictly speaking it should be absent.

> We introduce a new inheritance mode, which can be very useful to administrators who want a set of jails to behave the same.

When setting **`mac.do`** to **`inherit`**, **`mac_do(4)`** simply uses the configuration that applies to the parent jail, which itself may come from a jail higher in the tree. The main consequence is that a change of rules in any of the parent jails up to the first that does not inherit automatically and immediately does apply in a jail with **`inherit`**. This alleviates the administrator from having to change the configuration of multiple jails in a tree when all of them are supposed to stay in sync. As already noted, explicitly setting rules on a jail, whether through **`mac.do.rules`** or **`security.mac.do.rules`**, establishes independent per-jail settings, effectively breaking inheritance. Re-enabling it later is always possible, by just setting **`mac.do`** to **`inherit`** again.

As with any jail parameters, you can use these to easily configure a jail at its creation, either directly on **`jail(8)`**'s command-line, e.g.:

```
jail -c name=test_jail path=/ mac.do=inherit
```

or through **`jail.conf(5)`**. To modify some parameters as the jail is running, use **`jail -m`** as usual, e.g.:

```
jail -m name=test_jail mac.do=disable
```

Boot-up

Since **`mac_do(4)`** configuration on the host happens via **`sysctl(8)`** knobs that are also tunables, you can use either of the two different mechanisms that the base system provides to set them at boot.

As a first possibility, you can tune your **`loader.conf(5)`** configuration, by adding a line like:

```
security.mac.do.rules='uid=10001>uid=80,gid=80,+gid=80;uid=10001>uid=80'
```

This is adapted to cases where rules really have to be available very early at boot, or, e.g., if you are not using the base system's **`rc(8)`** bring-up framework.

Else, you can add the exact same line to **sysctl.conf(5)**, and the **sysctl(8)** knob will be set accordingly when **rc(8)** executes[21].

We have had some limited feedback that a few people do not find it very practical that **mac_do(4)** only deals with numerical IDs and find the **sysctl(8)** knob syntax quite terse. These are essentially the consequences of having the transition rules in the kernel, which allows to cope with a strong threat model where some userland parts may have been compromised. However, we understand that most people do not require such a level of security, and that having userland tools produce final rules from references to the content of the password and group databases could be useful to them. There has been a proposal in this direction consisting of a dedicated executable and configuration file for **mac_do(8)** which for the moment is stalled as, among others, we have been reflecting on the overall design, including how to organize executables and possible future configuration files and how to avoid conflicts. If more people are interested in such functionality, we may make progress on this front sooner rather than later. See also, in this article's last section, the short-term features we plan to add to **mdo(1)**, one of which will bridge an important part of the gap.

Some Notes on mac_do(4)'s Design

Baptiste Daroussin initially launched the **mac_do(4)**/**mdo(1)** project with the goal to enable role-based credentials transitions without using "setuid executables". In high-security, norm-constrained settings, installing these executables, if at all possible, may be subject to long and complex security audits, which often need to be renewed as the executables are upgraded. Thus, **mac_do(4)** was conceived as a kernel-based alternative that, thanks to the MAC framework[5], can authorize unprivileged processes to successfully change credentials. In addition to alleviating the need for "setuid executables", this architecture instantly reduces the impacts of a successful attack on or a programming bug of credentials-changing programs.

The original implementation of **mac_do(4)** only monitored the **setuid()** system call, authorizing a specific call to it according to rules matching the original user and the target one. In order to allow **mdo(1)** to change groups as prescribed by the password and group database for the target user, **mac_do(4)** then needed to accept any **setgroups()** and **setgid()** system calls. In order to avoid arbitrary programs from being able to leverage these calls independently, **mac_do(4)** would only authorize credentials transition requests from processes spawned from the **mdo(1)** program.

Because allowing any request from **setgroups()** and **setgid()** was a serious dent in reducing the impacts of an attack or a flaw, we modified **mac_do(4)** to validate the full credentials transition and its rules to say which groups can appear in the final credentials.

Validating or rejecting the full transition fundamentally requires atomicity, implying changes to the security API from traditional UNIX. A natural approach would be to add to the latter a transactional mode where successive calls amending credentials would not immediately apply changes but rather accumulate them for atomic application at final "commit". This approach would somehow facilitate amending existing programs as well as possible additions of credentials' attributes but was deemed relatively invasive in terms of kernel code and a paradigm change for the existing system calls' MAC hooks[22]. Instead, we settled for the alternative to have a new, separate system call, **setcred(2)**, that can set all credentials attributes at once. They are passed via a structure that can be extended or versioned through flags as needed. New MAC hooks are defined and are passed the current and requested credentials, allowing **mac_do(4)** to see the current and desired state at once and make decisions based on them.

Even after these changes, we have kept the restriction that `mac_do(4)` can only authorize processes spawned from the `mdo(1)` executable, as it may allow implementing additional transition restrictions in `mdo(1)` proper. A Google Summer of Code 2025 (GSoC 2025) student, Kushagra Srivastava, was tasked, among others, to bring configurability to this restriction, allowing an administrator to specify which executables `mac_do(4)` can authorize.

Some people may find strange, and even a potential security hazard, that code responsible for checking and deciding on credentials transitions based on rules is moved to the kernel instead of being executed in userland. While the ability to compromise the kernel would certainly be even more catastrophic than "setuid executables", we believe the former is much less likely to occur than the latter for the following reasons.

First, rules accepted by `mac_do(4)` are completely well-defined, self-contained, relatively simple to parse and hopefully to comprehend.

Second, "setuid executables" performing credentials changes generally involve a lot more components than `mac_do(4)` actually uses. The latter are essentially some parts of the MAC framework and the jail and OSD subsystems, which are pervasively used and tested and do not frequently or deeply change. The former are the libraries to read the password and group databases, which may involve network access, the userland configuration parser, and the code establishing all characteristics of the new session, including the credentials, which is sometimes part of a separate library, not even mentioning the usual userland support code, such as the dynamic linker.

"Setuid executables" performing credentials changes generally involve a lot more components than `mac_do(4)` actually uses.

Third, we have taken special care to design and write `mac_do(4)` in some of the cleanest and clearest ways, with special attention to understanding the constraints of the underlying subsystems and ensuring that the ones we rely on cannot be changed without our noticing via assertions. The result is that, despite copious testing, we have yet to find a bug in `mac_do(4)`'s core functionality (famous last words). Out of the few bug reports we received, only two turned out to be real problems in scenarios that admittedly were not well-considered nor tested initially[23], which led to performing another audit of the code. Our GSoC student was also tasked with developing automated tests, which should enter the official tree in the coming weeks. They will represent additional safeguards and will help maintain code quality as `mac_do(4)` and its dependent subsystems evolve.

What Lies Ahead

The essential message here is that, while we have a few simple short-term plans and more loose longer-term ones, future directions will depend for the most part on current or potential users' feedback. We are eager to hear suggestions for small improvements or entirely new features, whether you are already using `mac_do(4)`/`mdo(1)`, are planning to, or would like to but cannot because your use case is not covered by existing functionality. This will help us select what to work on while keeping the overall design sound. Even just saying you're using them is useful feedback, as it is good to know how many users we have and how they are using these tools.

In the short term, we expect to add auditing-like functionalities to `mac_do(4)`/`mdo(1)`. Displaying the final credentials passed to the kernel would help check if the invocation was correct with respect to the expected goals. Producing the target part of a `mac_do(4)`'s rule

authorizing exactly a specific `mdo(1)` call could help administrators build `mac_do(4)` configurations or better understand why some do not work as expected. Integration to the `audit(4)` subsystem would allow tracking credentials changes after the fact. Logging failed attempts through `syslog(3)` would match what `login(1)` and other credentials-changing program do. `mac_do(4)` will soon allow configuring the executables whose processes it will consider, with the aim to support thin-jails scenarios and other userland programs[24]. It should also monitor traditional system calls such as `setuid(2)` in addition to just `setcred(2)`, considering each call as a full transition on its own[24].

Longer term, we may consider providing `su`-like and `doas`-like functionalities, e.g., to ask for a password or perhaps more generally leveraging `pam(3)`, establish resource limits and other attributes as in a full login, or allow only certain commands to be launched. However, it is not yet clear how these functionalities could be fit into `mdo(1)`, as it is not a "setuid executable", and if different paths should be pursued instead.

Future directions will depend for the most part on current or potential users' feedback.

As an example, we have conducted a preliminary study on how to add support for requesting a password for certain credentials transitions. As `mdo(1)` can be launched by any user, we need a mechanism to check for a password against a password database which is not directly readable by everybody[25]. This situation is comparable to that of programs leveraging CAPSICUM's capability mode[26] which sometimes need to access data that require more privileges than they directly keep. That can be resolved by having an unrestricted process perform the necessary accesses on behalf of the process in capability mode. `libcasper(3)` is FreeBSD's implementation of that idea for a number of services, including `cap_pwd(3)` to access the password and group databases. Unfortunately, using `libcasper` as-is cannot work as `cap_enter()` creates and connects to a process launched with the same credentials. `mdo(1)` is going to need an outside daemon with privileges to provide the `cap_pwd(3)` service. We can also imagine a number of alternative approaches with varying development effort. They include pushing the password configuration entirely into `mac_do(4)` as for the rules, or turning `mdo(1)` into a "setuid" executable that however relinquishes root rights for most of its operation and crucially when calling `setcred(2)`, or instead leaving `mdo(1)` as it is and having a different "setuid" executable for these needs[27]. However, all of these alternatives except the first provide fewer security guarantees than the initial solution, and the first one is less flexible as it does not allow other forms of authentication nor additional transition restrictions that can be best imposed by userland[28].

We hope you will find `mac_do(4)`/`mdo(1)` useful! Please share your feedback and more generally other security needs you would like to see addressed, even if not necessarily directly connected to the framework presented here.

Footnotes

1. In reality, the special user ID 0. The name `root` resolves to ID 0, as may other names such as `toor`.
2. More precisely, the effective and saved user IDs, and the effective and saved group IDs, respectively. The saved user and group IDs are officially called the "Saved Set-User-ID" and "Saved Set-Group-ID" in the POSIX specification.
3. Other authentication mechanisms can be configured using PAM, see `pam(3)` for an introduction, `pam.conf(5)` for configuring particular applications, and `pam_unix(8)` for the canonical module.

4. As additional arguments to `su(1)` are passed to the target user's shell, the program and its arguments have to be passed through the shell's `-c` argument (or equivalent). For `sh(1)` and descendants, they must be grouped in a single argument that will be interpreted by the launched shell, sometimes requiring an additional level of quoting.
5. Mandatory Access Control. See `mac(4)`.
6. The updated `mdo(1)` described here will normally be shipped with FreeBSD 14.4.
7. There may be multiple users mapping to the same numerical ID. `doas(1)` has the flaw that it will silently consider the first matching user name. `mdo(1)` generally follows the conservative approach of not doing non-obvious operations silently, here not trying to use a matching user name, even if there is only one.
8. I.e., the real, effective and saved group IDs, by contrast with the supplementary groups.
9. To ease scripting, `-s` is actually compatible with `-G` and can be used to amend it, so it is in effect processed after `-G` even if it appears earlier on the command-line. Currently, though, using both `@` and `-G` is treated as an error (redundant specification), a limitation which may be lifted in the future.
10. The real user ID is matched, as it represents the user's identity, rather than the effective user ID, preventing by default another set of rules to apply for "setuid executables". That said, since unprivileged users are allowed to set the real user ID to the effective user ID on FreeBSD, this distinction is currently not an absolute restriction.
11. Since FreeBSD 15, a user's initial group from the password database is also installed as a supplementary group, which is also the case on Linux/glibc, NetBSD, OpenBSD, and illumos. For compatibility with FreeBSD 14.3, we demonstrate the target clause `+gid=80` here, which also works on 15.0, instead of `!gid=80`, which would allow the transition only on 15.0.
12. In more formal parlance, `gid` and `+gid` target clauses form a logical disjunction.
13. In more formal parlance, rules form a logical disjunction.
14. If `+gid=`. was replaced by `!gid=.`, the rule would allow a transition if and only if the current supplementary groups do not include 0, and not a transition to all current groups but 0. We may relax this constraint in the future.
15. There are some exceptions. We have seen one in the previous footnote. Another one is that, on one hand, the real, effective, and saved user IDs, and on the other hand, the real, effective, and saved group IDs are treated indifferently. Treating them separately was deemed to introduce additional complexity for meager benefit since FreeBSD's `setresuid()` currently allows an unprivileged process to set any of its user IDs to the value of any other one. We might want to disallow this behavior in the future.
16. Since FreeBSD 8.0.
17. Which always has a global jail ID of 0. Jail IDs are global, except that any process sees the ID of its immediately enclosing jail as 0.
18. In an earlier implementation, that parameter was called `mdo` and was intended to work like described here but did not due to bugs.
19. And thus, consistency issues.
20. Also known as Postel's Law. "Be liberal in what you accept, and conservative in what you send."
21. By the `/etc/rc.d/sysctl` script.
22. Either these hooks' existing implementations would need to start supporting the transactional mode, or we would bypass the hooks entirely, a change deemed too surprising to consumers.
23. Namely, using `mac_do(4)` when running with resource accounting functionality enabled, and running a 32-bit `mdo(1)` on a 64-bit architecture.
24. Most of the code for this functionality has been written during GSoC 2025 and should be integrated soon.
25. In order to avoid leaking password hashes that would allow offline attacks.
26. A process mode where most accesses to the global namespaces are restricted, and only existing file descriptors can be used.
27. That could take the form of first importing `doas(1)` into the base system and then tailoring it to our unique security features, although that would be a regression in terms of the granularity of target credentials. Alternatively, we could create an executable that would share part of its code and command-line interface with `mdo(1)`. Mixing both approaches to get the best of both worlds could also be viable.
28. But it has the benefit of not lessening the currently existing security guarantees, since the password would be checked by the kernel as well.
29. In the different real, effective and saved group IDs.

OLIVIER CERTNER has been continuously using FreeBSD on all his machines and those of some of the companies he worked with since the end of 2004. During this time, he has grown a set of private customizations including modifications to rc scripts and some kernel bits. After having worked for over 15 years in the CAD and finance sectors, he lately switched back to pure IT topics, and in particular operating system development. His main interests are centered around kernel development, with particular focuses on power management, security, scheduling, file systems and jails. He's currently a contractor for the FreeBSD Foundation.

printf("Hello, srcmgr\n");

BY MARK JOHNSTON

FreeBSD developers and devsummit attendees have likely heard of the newish srcmgr ("source manager") team at some point. But seeing as it's been a year or so since srcmgr started having regular calls, it seems time to introduce ourselves a bit more widely.

srcmgr is a team of src developers whose goal is to help organize FreeBSD src development. For context, FreeBSD's development model is somewhat unusual among OSS projects: rather than having an individual or small team of developers who direct the project and make high-stakes decisions, FreeBSD's src developers belong to a relatively flat hierarchy, subject, of course, to maintainership rules and conventions. Developers are collectively obliged to help push along src development, be it by finding or fixing bugs, writing documentation, reviewing code, adding features and tests, etc.

This development model works reasonably well — consider, for instance, that the FreeBSD project itself is older than some of our developers — but it has shortcomings too. Individual developers have few formal responsibilities; they are expected to contribute and work with each other, but there is little formal oversight. This model works well with small groups of developers who know and trust each other and can motivate one another; unsurprisingly, this characterized FreeBSD development during its early days. Over time, however, challenges arise. Long-time developers move on, the system grows more complex and harder to maintain, and the number of new developers increases, putting strain on experienced mentors, who typically have little free time.

FreeBSD's src developers belong to a relatively flat hierarchy

Historically, the FreeBSD Core Team provided fallback support: whenever there was a dispute or a problem with a neglected part of the tree, they would step in. This worked historically, as Core was for a long time mainly composed of src developers despite representing all of FreeBSD. But in the last couple of terms, the ports team has gained more representation within the Core team. Thus, recent Core teams have had fewer resources to devote to src-specific issues, and Core's attention should be focused on the project's long-term strategic direction rather than day-to-day matters.

Enter srcmgr. srcmgr was officially announced on the internal FreeBSD developer mailing list on October 8th, 2024, and currently consists of me, Ed Maste, Warner Losh, and John Baldwin. We also have five "lurkers," developers who attend srcmgr calls and participate without formally being srcmgr members; they aim to test the waters and decide whether they want to commit to becoming official members. At a glance, srcmgr plays the same role in the src tree as the portmgr and doceng teams do in the ports and doc trees, respectively: we try to provide oversight and help tackle challenges specific to our area.

Although the idea for a srcmgr team had been floated before, my first exposure came from conversations with John Baldwin and Ed Maste at BSDCan in 2023. It was recognized that Core tended to be overburdened and not have the capacity to do proactive work to

shepherd src development. Meanwhile, I was frustrated by our overall handling of bug reports, new contributors, and code reviews. As an individual developer, it was too much work to keep on top of everything while also doing regular paid work.

So, what does srcmgr do in practice? Well, our charter gives an outline. We meet every two weeks for roughly two hours; the first hour is spent reviewing agenda items, discussing them, and providing status updates. The second hour is typically spent triaging recent src bug reports and/or GitHub pull requests.

The primary responsibility of the group is to vote on src commit bits: when an src committer has been working with a contributor and believes that the contributor would make good use of src commit access themselves, they can send a proposal to srcmgr, who then votes on whether to grant the commit bit. This process tends to be uncontroversial and straightforward and consumes little time in practice.

We also spend time on "maintenance" tasks, such as disabling commit bits of inactive developers, pushing forward the deprecation of obsolete or unmaintained features that consume project resources, and updating developer policies and — with much help from FreeBSD's cluster admin team — bits of infrastructure such as git commit hooks.

The primary responsibility of the group is to vote on src commit bits

Most of our time is spent pushing along various initiatives, the principal aims being to 1) make experienced developers more productive, and 2) make it easier for newcomers to contribute. For the first goal, we have tried to push the creation of tools to make FreeBSD development easier. Today, it is far too difficult for new FreeBSD developers (interested contributors, GSoC students, employees at FreeBSD-using companies, etc.) to set up an environment where they can quickly and reliably test changes to the src tree. We have an extensive regression test suite, but actually running it requires a fair bit of setup and is tricky to automate. Setting up an efficient, interactive compile-edit-test loop is also tricky. Many developers have custom scripts and workflows to enable this, but that means that many developers end up reinventing the wheel; moreover, we do not have a good "canned" setup that newcomers can quickly adopt and customize. Solving this problem in general is a challenge, as FreeBSD is a large codebase with many components that require specialized development approaches. Still, there is a lot of room for improvement.

Following the theme of tooling, we have also been working on scripts to make MFCs easier and to programmatically catch certain classes of problems, such as missing backports on stable branches, particularly when commit B fixes a bug in commit A but is missed when merging commit A. Another area of focus is triaging incoming work for project members: bug reports, code review requests, and contributor patches. While individual FreeBSD developers spend a lot of time handling day-to-day requests of this nature, there is little oversight that ensures that high-priority issues don't fall through the cracks. srcmgr is composed of developers who are familiar with the src tree and can quickly identify who should be "tagged" on a particular issue to help move it forward.

Finally, one ongoing initiative is the hosting of "bug-busting" sessions, typically on Zoom or meet.freebsd.org. We announce these sessions in advance on the developer mailing list and invite folks to join us for roughly 3 hours of triaging and bug work. I typically lead these sessions by reviewing individual bug reports and looking for opportunities to make quick progress: assigning them to a subject matter expert, asking follow-up questions of the submitter, and discussing the problem with others on the call. People are free to participate as they see fit; some will quietly work on specific bugs in the background while keeping an ear on the chatter, and others will follow along or triage bugs in parallel.

These sessions (and srcmgr participation in general) do a lot to keep me motivated: real-time interaction with other developers helps keep up engagement for most of us who work remotely, and being able to make fast, tangible progress on bugs and pull requests keeps the backlog from feeling overwhelming. It's much easier to take pride in the project when we can stay on top of incoming issues and work requests, and this collaboration helps maintain our sense of shared responsibility.

There is a lot more srcmgr would like to do to help the project, and we have a lot of initiatives that we will be pushing along in the coming year, especially as the FreeBSD 15.0 release date edges closer and we get some time to recharge over the holidays.

If you have any feedback or ideas, please always feel free to email us at: srcmgr@FreeBSD.org.

MARK JOHNSTON is a FreeBSD developer living in Toronto, Ontario, Canada. When not sitting at a computer, he enjoys playing in a city dodgeball league with friends.

FreeBSD and Google Summer of Code 2025

BY JOE MINGRONE

The successful completion of Google Summer of Code (GSoC) 2025 marks FreeBSD's 21st consecutive year participating in the program. Three factors made this year stand out. First, we received 64 applications, which is more than double last year's total and roughly four times the number received in 2023. AI tools likely contributed to the surge, producing some low-quality submissions; however, the overall quality of most applications remained high. Second, we saw a notable increase in interest from South Asia. Given that the FreeBSD Community Survey results indicate that 85% of respondents were from Europe or North America, this interest from Asia and other parts of the world is welcome. Third, the number and quality of accepted projects were encouraging. Out of 1200 total GSoC projects across 185 participating organizations, FreeBSD's 12 accepted projects nearly doubled the average per organization, and unlike last year, all 12 were successfully completed.

Before we discuss individual projects, let's reflect on why we participate in GSoC. The program requires significant effort, from organizing the application process and defining project ideas to mentoring contributors. Is this investment of time and resources, which could otherwise go toward direct development, worthwhile? From a short-term technical perspective, it's debatable; while some projects lead to committed code, many do not. However, considered from a long-term perspective, the answer is clearer. GSoC is playing an important role in attracting and developing new contributors. Since 2022, five new FreeBSD committers have come through GSoC, and one 2017 participant went on to serve on the 12th Core Team.

GSoC 2025 Projects

Sockstat UI Improvements

For those unfamiliar with **`sockstat(1)`**, it is a command to list open Internet or Unix domain sockets. Damin Rido's goal for this project was to enhance the flexibility of the command's output by allowing dynamically sized columns and integrating libxo for structured output support. All three of Damin's pull requests linked below were pushed to the **`src`** tree.

- Add automatic column sizing and remove -w option: freebsd/freebsd-src#1720
- Reintroduce **`-w`** flag to automatically size the columns: freebsd/freebsd-src#1746
- Add libxo support: freebsd/freebsd-src#1770

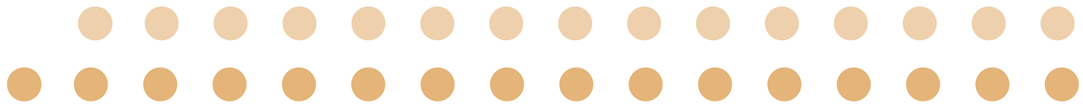
`vmm(4)` Accelerator Support for QEMU

The VMM (Virtual Machine Monitor) module is the kernel component of the **`bhyve(4)`** hypervisor, accessible through **`vmm(4)`**. Other than bhyve, another widely used machine emulator and virtualizer with official support for FreeBSD is QEMU. However, prior to this project, QEMU on FreeBSD could only make use of software emulation (via its Tiny Code Generator) because it lacked support for hardware-accelerated virtualization. In other words,

QEMU on FreeBSD could not take advantage of the host CPU's virtualization extensions to run guest code directly on the hardware. This limitation resulted in significantly higher CPU overhead and slower guest performance compared to hardware-assisted virtualization.

The primary objective of Abhinav Chavali's GSoC 2025 project was to integrate VMM acceleration support into QEMU on FreeBSD. He accomplished this by modifying QEMU's memory management layer to interoperate with VMM's kernel-allocated guest memory. He also adapted VMM to make certain non-critical devices such as the HPET and RTC optional, allowing QEMU to emulate them in user space instead. This enables a hybrid interrupt model (with the virtual LAPIC handled in the kernel and the IOAPIC emulated in user space), which has the potential to deliver performance levels comparable to bhyve under FreeBSD.

The latest from from Abhinav is that he was able to successfully boot FreeBSD 14 under QEMU with the VMM acceleration. You can view his code here.



The latest from Abhinav is that he successfully booted FreeBSD 14 under QEMU with VMM acceleration.

Testing and Development for Rust FreeBSD Device Drivers

Over the past few years, there has been interest in incorporating Rust in both FreeBSD and Linux development. For a sample of some discussions, refer to an RFC for Rust support that was posted to the Linux Kernel mailing and to discussions on the FreeBSD hackers list. In both communities, debates have emerged: some opponents warned of issues such as "doubling build times", whereas proponents argued that Rust would make certain tools easier or even possible to implement. Beyond discussions, tangible progress has been made. In his Master's thesis project, Johannes Lundberg created a Rust KPI and network driver, while David Young created a simple "Hello World" FreeBSD kernel module in Rust and summarized existing community efforts to adopt Rust.

The *Testing and Development for Rust FreeBSD Device Drivers* project builds on past efforts to incorporate Rust into FreeBSD development. One of its primary goals was to create a testing and continuous-integration framework for Rust kernel modules. Aaron gives an overview of his Rust echo driver in this video and summarizes the project in this write-up. His code is available here:

- https://github.com/Acesp25/rustdrv
- https://github.com/Acesp25/freebsd-kernel-module-rust
- https://github.com/Acesp25/RustKLD

Full-Disk Administration Tool for FreeBSD

Prior to this GSoC 2025 project from Braulio Rivas, FreeBSD lacked a user-friendly tool for full-disk administration, i.e., a utility comparable to Linux's GParted for partitioning, resizing, moving, and managing file systems. The goal of this project was to fill that gap by creating a new partitioning tool called geomman. Upon completion of the project, geomman supports the following operations:

- copy and paste partitions on the same disk or across disks
- grow UFS, NTFS, ext2, ext3 and ext4 filesystems
- shrink NTFS, ext2, ext3 and ext4 filesystems

- visually select free space to place a partition
- create exFAT, NTFS, ext2, ext3 and ext4 filesystems
- check filesystems: fsck_ufs (UFS), fsck_msdos (FAT), fsck.exfat (exFAT), ntfsfix (NTFS), and e2fsck (ext)
- create and label a partition
- create and encrypt a partition

Remaining work:

- ZFS support
- resolve issues when moving a partition
- test cases

The upstream repository can be found at: https://gitlab.com/brauliorivas/geomman.

Adding QCOW2 Compressed Image Support to mkimg

QCOW2 (QEMU Copy-On-Write version 2) is a widely used disk image format for virtualization, recognized for features such as thin provisioning and built-in compression. FreeBSD's `mkimg(1)` tool can create disk images in a variety of formats, including QCOW2. Until now, however, mkimg's QCOW2 support was limited and did not allow for the creation of compressed QCOW2 images.

This summer, Christos Komis enhanced mkimg by completing these milestones:

- add support for QCOW2 v2 compressed images
- add support for QCOW2 v3 compressed and uncompressed images
- update the user interface to expose the new features
- extend the test suite to verify correctness
- update the man pages with the new functionality
- perform code refactoring to improve readability and maintainability.

The implementation has been thoroughly tested and is ready for commit. Users can now generate compressed QCOW2 images directly with mkimg, simplifying workflows for virtual machine image generation and reducing reliance on external conversion tools. Check out Christos's code at https://github.com/ckkomis/freebsd-src/commits/mkimg/qcow2-compression/.

ACPI Initialization in Loader With Lua Bindings

Kayla Powell's project extends the ACPICA library's initialization into the FreeBSD loader, ensuring the full ACPI namespace is available before the kernel is loaded. The work replaces a somewhat ad hoc bootloader approach to ACPI by invoking standard ACPICA routines (e.g., AcpiInitializeSubsystem, AcpiLoadTables, AcpiWalkNamespace, AcpiEvaluateObject) within the loader. This gives consistent discovery and interrogation of ACPI objects early in the boot process. To maintain the loader's lightweight design, only the necessary ACPICA components were ported, omitting many functions unnecessary for initialization or scripting.

On top of the foundational layer, the project introduces Lua bindings that expose the ACPI namespace and object-evaluation facilities to scripts running under the loader. In other words, we can now write Lua loader code to walk the ACPI tree, examine device-table entries, and attach or read data from ACPI nodes, before the kernel loads. Along with the implementation, unit and regression tests were included (e.g., comparing namespace dumps between C and Lua and building the loader across architectures).

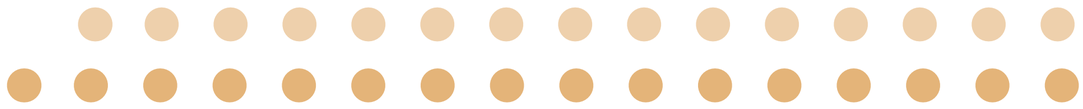
Refer to Kayla's summary of the project on her blog at https://kmpow.com/content/gsoc-writeup and her pull requests: 1818, 1819, and 1843.

`mac_do(4)` and `mdo(1)` Improvements

Kushagra Srivastava's project aims to enhance FreeBSD's credential transition infrastructure by improving both the kernel-side MAC module, **`mac_do(4)`**, and its companion userland tool, **`mdo(1)`**. Rather than relying on traditional setuid executables (which carry inherent risks), the goal is to enable controlled, fine-grained credential transitions under the umbrella of FreeBSD's MAC framework. On the kernel side, **`mac_do(4)`** was extended to support per-jail configuration of authorized executables (so that an admin can specify exactly which binaries in a given jail are allowed to request credential transitions, instead of being limited to a hardcoded path). Also, it now intercepts standard credential-changing syscalls such as **`setuid(2)`**, **`setgid(2)`**, **`setgroups(2)`**, and treats them as full transition requests that are subject to the **`mac_do(4)`** policy module.

For userland work, the **`mdo(1)`** tool was improved to provide fine-grained credential transition requests. It now supports explicit overrides of user/group IDs, as well as supplementary groups, via flags such as **`-g`**, **`-G`**, and **`-s`**. It also includes a **`--print-rule`** option to display the matching **`mac_do(4)`** rule for a requested transition, which helps administrators with rule creation and debugging.

Together, these enhancements make credential transitioning more flexible, secure, and integrated with FreeBSD's jail and MAC frameworks. This reduces the need for risky setuid binaries and brings improved auditability and control.



These enhancements make credential transitioning more flexible, secure, and integrated with FreeBSD's jail and MAC frameworks.

Refer to Kushagra's project write-up at https://thesynthax.hashnode.dev/my-google-summer-of-code-journey-part-3 for details.

Speed up the FreeBSD Boot Process

Lahiru Gunathilake's project is a continuation of past projects to speed up FreeBSD's boot time by profiling the boot sequence, identifying bottlenecks, and implementing optimizations. Using the built-in TSLOG tracing framework, Lahiru generated flame charts of the boot path to understand where time was being spent and where unnecessary delays could be eliminated.

Once the profiling revealed hotspots such as device attach phases, initialization of large filesystem subsystems (especially ZFS), and sleeps in vfs_mountroot (root filesystem mount), the work progressed to the implementation phase. This included:

- reducing a benchmarking buffer size from 16MB to 256KB, cutting startup from 989 ms to 67 ms
- reducing long wait loops in keyboard and mouse initialization, and introducing the tunable hw.atkbd.short_delay
- eliminated unnecessary waits for USB devices.

Overall, Lahiru reported reductions of 8.2 s in kernel initialization, 3.5 s after the ZFS and input device optimizations, and 1.9 s when skipping the USB boot wait.

WiFi Management UI

Muhammad Saheed took on a project to develop cohesive CLI (wutil) and TUI (wutui) utilities for managing WiFi networks on FreeBSD. The aim was to cover "station mode op-

erations, such as scanning, connecting/disconnecting from wireless networks," and to wrap these into a clearer, more consistent user interface. Other completed work includes:

- updating related man pages
- creating a port for wutil
- adding libwpa_client build option to security/wpa_supplicant port
- creating a port for libifconfig
- extracting required ifconfig helpers into libifconfig (D52130, D52131)

Refer to Muhammad's blog for more information about his work.

Journaling for FreeBSD ExtFS

This project by Pau Sum set out to bring Linux-compatible journaling to FreeBSD's ext2fs filesystem implementation. With FreeBSD's existing ext2fs driver, FreeBSD users could already mount and use ext2/3/4 filesystems, but the driver lacked journaling support, meaning unclean shutdowns required lengthy recovery via fsck. Pau's work introduced on-disk journal awareness and transaction logging to improve crash recovery and filesystem integrity, allowing FreeBSD to mount and replay journals on ext3/4 volumes and interoperate more closely with Linux systems.

Rather than replicating Linux's full journaling framework, the design implements a traditional metadata-only journal using the same on-disk structures for compatibility. The new code defines key data structures, including ext2fs_journal (representing the active journal), ext2fs_journal_transaction (grouping atomic metadata updates), and ext2fs_journal_buf (tracking per-block state). Core filesystem operations like ext2_link, ext2_mkdir, and ext2_write were extended with journal hooks to begin, dirty, and end transactions. Committing a transaction writes descriptor, metadata, revoke, and commit blocks, followed by checkpointing to flush updates to disk. Recovery proceeds in three passes: validating transaction ranges, collecting revoked blocks, and replaying non-revoked metadata.

By the end of the project, 11 of 12 journal hooks were complete, with work in progress on truncation and extent-based operations. Planned extensions include journaling support for extents and truncation, checksum validation for journal integrity, more extensive crash simulation, and documentation cleanup. The implementation was tested using fsx and dirconc. Pau's code is available from his fork of FreeBSD's src repository.

Power Profiling Tool

The goal of Kasyap Jannabhatla's project was to provide granular, process-level insights into power usage on FreeBSD. This addressed the limitations of ACPI's whole-system power statistics. Inspired by Performance Co-Pilot (PCP) and RAPL (Running Average Power Limit) support, the project implemented a FreeBSD-native framework rather than porting Linux PowerTop. The solution consisted of a kernel-level component to collect power-related metrics and a user-space daemon with a command-line interface that provides CPU usage and energy consumption per process tracking. By combining RAPL readings with per-process CPU utilization derived from kvm_getprocs, the tool can estimate energy usage for individual processes and threads, providing a foundation for fine-grained power profiling and future enhancements in FreeBSD's power management ecosystem.

Throughout the development, the project focused on building a lightweight daemon-based architecture, implementing a library (librapl) for structured RAPL data access, and integrating it with process accounting to calculate per-thread energy consumption. Testing involved stress benchmarks such as OpenSSL Speed and careful handling of multi-

core runtime accounting using thread IDs. By the end of the project, the framework could reliably report per-process energy usage over time, and all deliverables, including the daemon, library, and documentation, were completed. The implementation is available here.

Port FreeBSD to QEMU microvm

QEMU microvm is a minimalist virtual machine inspired by Firecracker. While FreeBSD was ported to Firecracker, that platform is Linux-specific, limiting portability. For this project, Wyatt Geckle aimed to develop a QEMU microvm version of the FreeBSD kernel inspired by the Firecracker port. To do this, Wyatt replicated prior porting attempts and analyzed kernel initialization issues, particularly timer configuration, which caused long boot times. By studying NetBSD's working microvm implementation and Intel documentation, the project identified differences in FreeBSD's timecounter and APIC initialization.

The current FreeBSD microvm kernel boots under QEMU microvm but does not call tc_init, which limits the available timers. The Firecracker port remains broken due to MPTables issues, requiring further investigation. Despite these limitations, the project broadened Wyatt's understanding of FreeBSD and NetBSD kernel internals, virtualization, and microvm platforms, and produced extensive documentation for building, running, and debugging FreeBSD in microvms and Firecracker. The work also provides a foundation for future contributions to FreeBSD, QEMU microvm, and Firecracker support, as well as reproducible debugging workflows for other microvm projects. Those interested in this work can find more information on Wyatt's blog. The code can be found at Wyatt's fork of the src tree.

It's gratifying to review the success of our GSoC 2025 program, but the time before GSoC 2026 starts will come quickly.

Mentor Summit

Robert Clausecker, a FreeBSD GSoC co-administrator and mentor, represented FreeBSD at this year's Mentor Summit that was held from October 23 to 25 in Munich, Germany. Topics discussed included AI-generated and spam applications. While no definitive solutions have emerged, one approach under consideration is to require applicants to meet with potential mentors before applying. This is something FreeBSD has already encouraged in previous years to help ensure good matches between mentors, contributors, and projects. Robert also met with representatives from the Linux Foundation to brainstorm potential collaboration between the Linux and FreeBSD Foundations, such as attracting more students to operating systems development.

Other summit sessions covered funding in open source projects. Robert spoke with a developer working on RTEMS, a real-time operating system used in various devices and learned that they incorporate FreeBSD's network stack in their system. He also met GSoC organizer Stephanie Taylor and shared the positive impact GSoC has had on FreeBSD. Of course, he returned with some swag, including a T-shirt and a pair of GSoC(k)s.

Final Thoughts

It's gratifying to review the success of our GSoC 2025 program, but the time before GSoC 2026 starts will come quickly. As usual, our most significant challenges to repeating this year's success will be developing suitable projects, finding dedicated mentors, and matching applicants to mentors. Fortunately, recent changes to the Google Summer of Code program should help.

- Flexible Timelines and Scope: Project timelines are more flexible. Contributors and mentors can choose from small (90-hour), medium (175-hour), or large (350-hour) projects, and the total time for the projects can be extended from the standard 8 weeks (small) or 12 weeks (medium and large).
- Expanded Contributor Pool: The pool of applicants has grown. Contributors do not have to be university students, so everyone new to open source is eligible to participate.

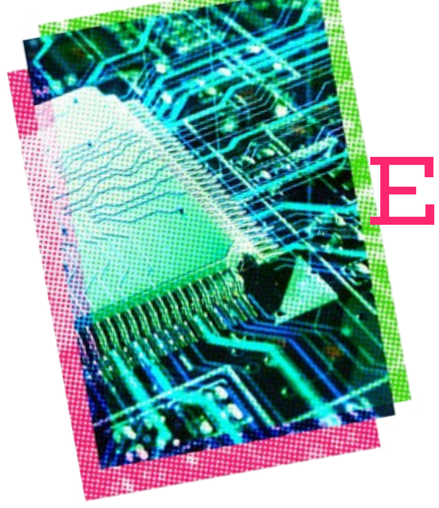
But for now, let's bask in our collective success and acknowledge the considerable effort that went into this year's program.

Thank you

Thank you to everyone who contributed project ideas to https://wiki.freebsd.org/SummerOfCodeIdeas and especially to our 2025 mentors:

- John Baldwin
- Olivier Certner
- Robert Clausecker
- Pedro Giffuni
- Tom Jones
- Warner Losh
- Ed Maste
- Getz Mikalsen
- Joe Mingrone
- Mehdi Mokhtari
- George Neville-Neil
- Colin Percival
- Alfonso Sabato Siciliano
- Alan Somers
- Toomas Soome
- Fedor Uporov
- Aymeric Wibo

JOE MINGRONE is a FreeBSD ports developer and works for the FreeBSD Foundation. He lives with his wife and two cats in Dartmouth, Nova Scotia, Canada.



Embedded FreeBSD

Building U-boot

BY CHRISTOPHER R. BOWMAN

A reader wrote to me that he had trouble building U-boot, so I thought I'd walk through the process since I wanted to bring up a different Zynq board and would have to go through it anyway. I need to provide a disclaimer: what I've written below is accurate, but these are complex systems, and I could have gotten some details wrong. I'd be grateful to hear from you if you think I have.

As we've discussed before, U-boot is both the second- and third-stage boot loader that runs and loads the FreeBSD loader, which, in turn, loads the FreeBSD kernel itself. U-boot is an open-source community project used on a wide variety of systems to provide boot services. Documentation is available through their website: The U-boot Documentation.

U-boot is both the second- and third-stage boot loader that runs and loads the FreeBSD loader.

On AMD/Xilinx Zynq chips, the first-stage boot loader is in BootROM on the Zynq chip itself. The Zynq boot process is described in Chapter 6, "Boot and Configuration" of the Zynq 7000 SoC Technical Reference Manual. The short description is that when powering on the device, it samples some pins and, depending on their state, chooses one of several boot methods. This allows a jumper on the board (JP5 according to Fig. 2.1 of the Zybo Z7 Reference Manual) to select a boot method, one of which is loading from an SD card. If you select this method, the BootROM code looks for a file named **`boot.bin`** on a FAT16 or FAT32 partition on the SD card. This is what U-boot calls the secondary program loader or SPL. The Zynq chip contains a small amount of onboard RAM, thus limiting the size of the program that the BootROM can load. In non bare metal applications, the SPL must contain enough code to configure the Zynq chip (PLL's, memory interfaces, and more) so that it can bring up the memory system and load U-boot proper. U-boot proper is a larger version with more features (like file systems) supported.

We're quite lucky as the Zybo Z7 is already supported in U-boot. So, we just need to get the build process to work. U-boot is typically built on a host system for a target system. The U-boot documentation referenced above suggests that we should set the compiler to be used for this cross-compiling using an environment variable. We will use the GCC compiler and GNU tools to do the cross-compiling, so we need to install those packages and set the cross compiler. U-boot is also built using **`gmake`**, not the standard BSD make, so we'll need to install that, as well as some other packages:

```
pkg install gmake
pkg install arm-none-eabi-gcc
pkg install bison
pkg install gnutls
pkg install gmake
pkg install pkgconf
pkg install coreutils
pkg install dtc
pkg install gdd
```

```
setenv CROSS_COMPILE arm-none-eabi-
```

Odd that you use **arm-none-eabi-**, and not **arm-none-eabi-gcc**, but it's not a typo.

Next, we need to configure the U-boot source tree for the board we want to target. The Zybo Z7 board is supported by the **xilinx_zynq_virt_defconfig** located in the **configs** directory. This configuration supports multiple boards, one of which is the Zybo Z7. To configure the source tree, we run:

```
make xilinx_zynq_virt_defconfig
```

But we have to be careful that we pull in the GNU **make**, not the BSD **make**. To do this, I've created a directory with a symlink named **make** that points to **/usr/local/bin/gmake**, and I've set this directory to be first in my path. This seems to work well. From there, we can just call **make** and wait (I highly recommend using the -j flag if you have extra cores). Did it error out for you as it did for me?

I get this output:

```
make[1]: *** [scripts/Makefile.xpl:257: spl/U-boot-spl-align.bin] Error 1
make: *** [Makefile:2358: spl/U-boot-spl] Error 2
make: *** Deleting file 'spl/U-boot-spl'
```

The relevant lines from **scripts/Makefile.xpl** are

```
$(obj)/$(SPL_BIN)-align.bin: $(obj)/$(SPL_BIN).bin
        @dd if=$< of=$@ conv=block,sync bs=4 2>/dev/null;
```

If you remove the redirection of output to **/dev/null**, you'll see a complaint from dd:

```
dd: record operations require cbs
```

Seems FreeBSD's **dd** is not command line equivalent with the GNU version. Originally, I simply used the GNU version of **dd** by installing the package and then creating a symlink in my local bin directory, but it turns out you can simply remove "block" from the **dd** command.

Also, the **V make** variable can be set to control the verbosity of build output. If your build doesn't work, I highly recommend running again with only one processor and V=1:

```
make V=1
```

If everything builds without error, you should have a **U-boot.img** file and an **spl/boot.bin** file. These are U-boot proper and the secondary program loader. Copy these to your SD card and give it a whirl!

Wha, wait, didn't work? Huh! As I said, this configuration supports multiple boards, and its default device tree isn't for the Zybo Z7. Consulting the board-specific documentation referenced above, we can specify which device tree is the default by setting DEVICE_TREE:

```
setenv DEVICE_TREE zynq-zybo-z7
```

This will override the default DTS in the configuration file. Build it again and try it. Wait, what? Another problem? The kernel loads, but it crashes in probing? Oh right. FreeBSD DTS requirements are not the same as Linux. The compat strings required to get some hardware recognized are different, and FreeBSD seems to require some clock-frequency properties, though I'm not sure the values are used. It might make sense to add compat values to the FreeBSD drivers that match what Linux expects, but I'm not a committer. I had to add the following to the DTS file in **arch/arm/dts/zynq-zybo-z7.dts**:

```
&sdhci0 {
        compatible = "arasan,sdhci-8.9a", "xlnx,zy7_sdhci";
   U-boot,dm-pre-reloc;
        status = "okay";
};

&devcfg {
compatible = "xlnx,zynq-devcfg-1.0", "xlnx,zy7_devcfg";
status = "okay";
};

&global_timer {clock-frequency = <50000000>;};
&ttc0 {clock-frequency = <50000000>;};
&ttc1 {clock-frequency = <50000000>;};
&scutimer {clock-frequency = <50000000>;};
```

Now that we've learned to build U-boot, let's see if we can make it a port. There are a whole bunch of U-boot ports, all of which are built off the U-boot-master port. To use them, we need to include the master port **Makefile**. We have to specify the board, the model, and the config that should be used. We have a few patches for the changes we made above, and we end up with the following.

```
MASTERDIR=      ${.CURDIR}/../U-boot-master

MODEL=          zybo-z7
BOARD_CONFIG=   xilinx_zynq_virt_defconfig
FAMILY=         zynq_7000

EXTRA_PATCHES=  ${.CURDIR}/files

BUILD_DEPENDS+= gdd:sysutils/coreutils

COMMENT=        ported by Christopher R. Bowman <my_initials>@ChrisBowman.com

.include "${MASTERDIR}/Makefile"
```

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 two 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 hard-ware design and the software that drives it. He has worked in the semiconductor design automation industry for the last 20 years.

2026 Events Calendar

BSD Events taking place through March 2026

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.

Code and Compliance FOSDEM Edition

January 26, 2026
Brussels, Belgium
https://www.eclipse-foundation.events/event/code-compliance-2026/

Join us in Brussels for the next Code & Compliance gathering. In this open, community-driven event, open source developers, project maintainers, and industry leaders come together to delve into the EU Cyber Resilience Act, enhance open source compliance practices, and share practical approaches to improving software security.

FOSDEM 2026

January 31 - February 1, 2026
Brussels, Belgium
https://fosdem.org/2026/

FOSDEM is a two-day event organized by volunteers to promote the widespread use of free and open source software. Taking place on January 31 & February 2026, FOSDEM offers open source and free software developers a place to meet, share ideas, and collaborate. Renowned for being highly developer-oriented, the event brings together some 8000+ developers from all over the world. This year, there will also be a BSD Dev Room and Stand.

SCALE 23X

March 5-8, 2026
Pasadena, CA
https://www.socallinuxexpo.org/scale/23x

SCaLE 23X – the 23rd annual Southern California Linux Expo is the largest community-run open-source and free software conference in North America. It is held annually in the greater Los Angeles area. The Foundation will be exhibiting this year.

AsiaBSDCon 2026

March 19-22, 2026
Taipei, Taiwan
https://2026.asiabsdcon.org/

AsiaBSDCon is a conference for users and developers on BSD based systems. It is a technical conference and aims to collect the best technical papers and presentations available to ensure that the latest developments in our open-source community are shared with the widest possible audience.