FreeBSD® JOURNAL

January/February/March 2026

Consolations for Kernel Hackers

Let Sleeping CPUs Lie – S0ix

How the Foundation's Laptop Support & Usability Project Came Together

Adventures in TCPIP: Improving TCP's Responses to Reordering

Duplicating your System

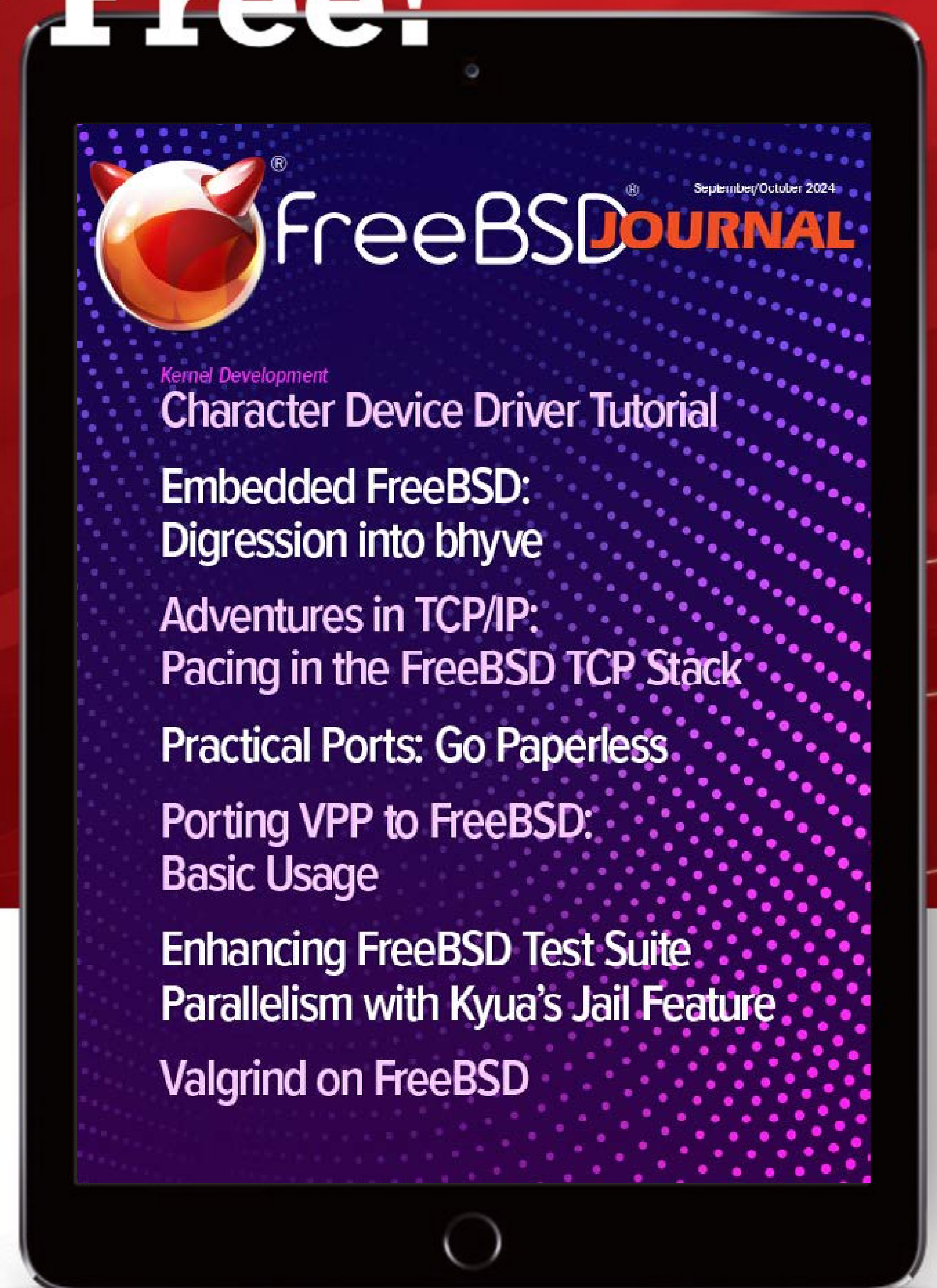
FreeBSD® JOURNAL

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

Editorial Board

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

Tom Jones • FreeBSD Developer, Software Engineer, FreeBSD Foundation

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

Benedict Reuschling • FreeBSD Documentation Committer

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

Mariusz Zaborski • FreeBSD Developer

Advisory Board

Anne Dickison • Deputy Director FreeBSD Foundation

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

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

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

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

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

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

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

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

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

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

Design & Production • Reuter & Associates

FreeBSD Journal (ISBN: 978-0-61 5-88479-0) is published 4 times a year (January/February/March, April/May/June, July/August/September, October/November/December).
Published by the FreeBSD Foundation,
3980 Broadway St. STE #103-107, Boulder, CO 80304
ph: 720/207-51 42 • fax: 720/222-2350
email: info@freebsdfoundation.org

Copyright © 2026 by FreeBSD Foundation. All rights reserved.
This magazine may not be reproduced in whole or in part without written permission from the publisher.

LETTER from the Foundation

Dear *FreeBSD Journal* Community,

I'm excited to share our latest issue of the *FreeBSD Journal*, highlighting FreeBSD on modern desktops and laptops, which has been an ongoing priority for the Foundation.

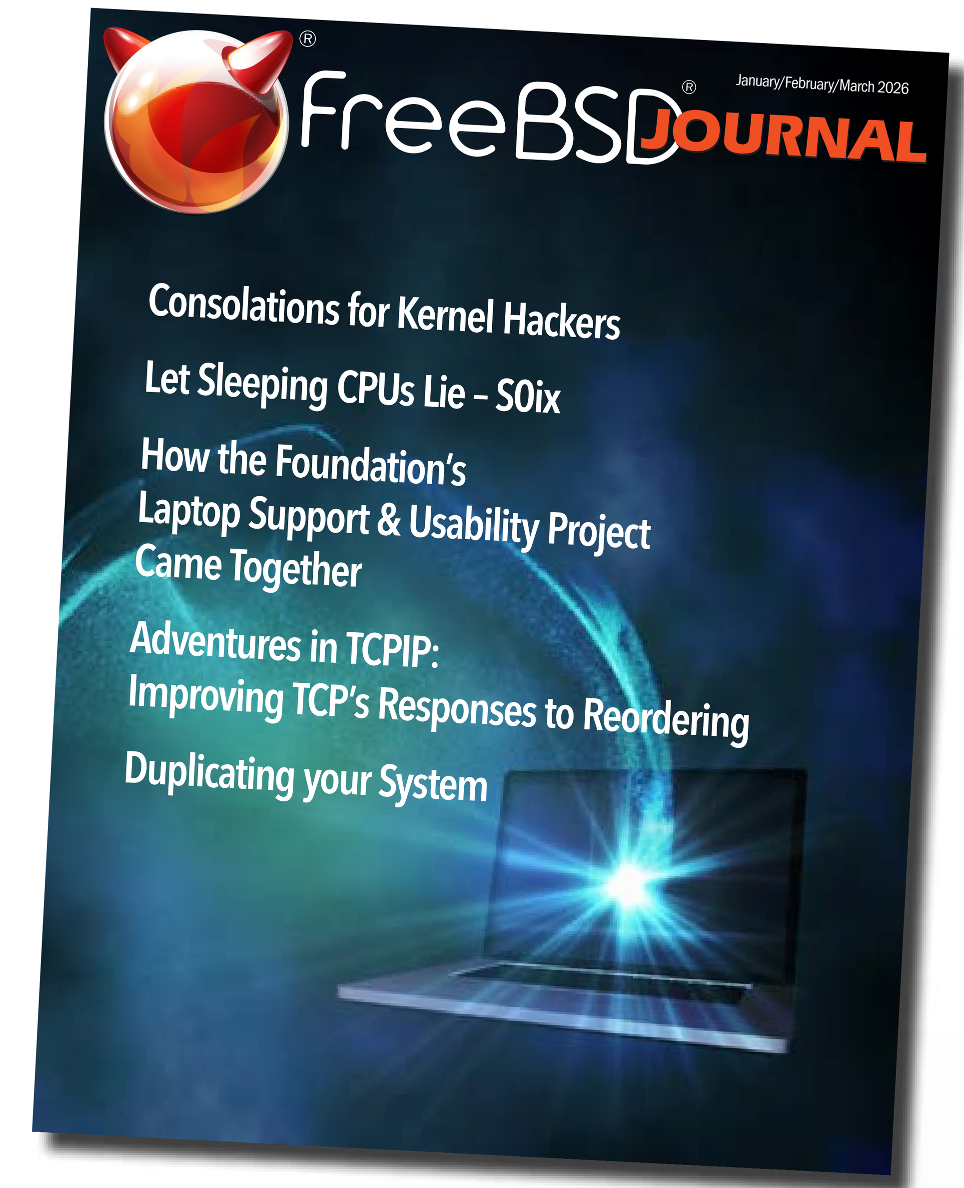
Over the years, the FreeBSD Foundation has consistently invested in improving the FreeBSD desktop experience, with a goal of making it more accessible to a broader community of users. With the support of targeted funding, as mentioned in my article, we've recently accelerated our efforts in this area. A few articles in this issue provide a closer look at some of the work on the laptop project.

Inside this issue, you'll learn about improving TCP responses to reordering, what a console driver is, recent improvements to power management, and how to duplicate your desktop. I also share insights into how our laptop support and usability project came together.

Happy Reading!
Deb Goodkin
Executive Director of the FreeBSD Foundation

January/February/March 2026

FreeBSD® JOURNAL



Topic: Laptop/Desktop

8 **Consolations for Kernel Hackers**
By Tom Jones

23 **Let Sleeping CPUs Lie — S0ix**
By Aymeric Wibo

30 **How the Foundation's Laptop Support & Usability Project Came Together**
By Deb Goodkin

34 **Duplicating your System**
By Jason Tubnor

3 **Foundation Letter**
By Deb Goodkin

5 **We Get Letters**
By Michael W. Lucas

40 **Adventures in TCPIP: Improving TCP's Responses to Reordering**
By Randall Stewart

45 **Events Calendar**
By Anne Dickison

WeGet letters

by Michael W Lucas

Dear Person Who's Been Running FreeBSD for Longer than I've Been Alive,

Commercial operating systems are increasingly intrusive and exploitative. I want a better life. How do I get started with desktop BSD?

—Avoiding Wasting One's Life

Oh AWOL, my succulent summer child,

You're the latest model of Human™, a digital native, theoretically enlightened by access to the collected wisdom of our species. Are your sources of wholesome, cogent, and actionable advice so limited that reaching out to a grizzled sysadmin seemed wise? True, this is an advice column. I offer advice. I also offer trepanning and experimental epidemiology, but you don't see people queuing for either of those, do you? The Internet has Captain Awkward and Doctor Nerdlove and Dear Prudence and Ask A Manager and Dear Abby, and you decided that no, you want to live abhorrently, so you're beseeching a grant of "wisdom" from a system administrator known for enhancing rat-operated vehicles with nitrous oxide boosters. If you're from a civilized country you even have health care so you could access counseling, or perhaps I should say intensive counseling because a few minutes into your first session the therapist would hit the big red button and you'd have a comfortable new home with three hot meals a day, your own bed guaranteed 98% free from arthropod infestation, and a meticulously personalized medication regimen because, after all, you think that sysadmins dispense sensible advice for a better life. Instead, you've requested guidance from a human(ish) creature who works on a split keyboard mounted at his hips because the sheer quantity of the computing he has perpetrated has wrought irreversible physiological changes in his meatsuit.

> **Are your sources of wholesome, cogent, and actionable advice so limited that reaching out to a grizzled sysadmin seemed wise?**

Very well. You want a better life.

Abandon computers. Learn subsistence farming, homebrew medicine, and chicken breeding. These will all be useful after the Big Crash. It's hard to say when the Big Crash is coming because even the combined efforts of every sysadmin on the planet with over two years of experience resist scheduling, though we're a stubborn lot, so we shall unquestion-

ably complete the Great Work, but a few of us are using AI to accelerate the race to Permanent Downtime, so might I suggest you hurry? AI might be sewage, but if you seek downtime, it's unmatched.

Still here? Fine. You came to my advice column, so I must offer advice.

I advise suffering.

Get a clean, minimal install, without any fancy desktop environment. Get a console-based web client like lynx(1). That will get you access to mailing list archives and support forums without supporting videos, social media, or any of those time-leeching parasites. You want to play video games? Learn how to get a graphic console working. Or play Nethack.

It's uncomfortable. The discomfort is the point.

Pain is the greatest teacher, but nobody willingly attends her classes.

Snuggle the pain.

Living with a Unix desktop is much easier than it once was. Window managers like KDE and Gnome offer a pointy-clicky experience indistinguishable from the operating systems you purport to flee. Those are for people who want the commercial OS experience without the exploitation, which is rather like wanting rain without the damp. Starting with a classic window manager like FVWM will broaden your awareness of what's possible. Window managers like OpenBSD's cwm will liberate you from the mouse, leading you to understand just how much time you waste reaching for the rodent and plunging you down the mountain towards irreversible physiological change in your own meatsuit.

Pain is the greatest teacher, but nobody willingly attends her classes.

After using different window managers for a few weeks, you'll wonder how you ever endured the pointy-clicky commercial wasteland.

Remember that as you approach other tasks. The narrow options presented by commercial operating systems are rails. They allow you to go one way. When you leave the rails and go by foot, you will discover a world overflowing with software written for innumerable kinds of brains. Try one program, then another. Yes, trying things is a pain.

You want a graphical mail client?

You want to watch DRM-protected streaming media?

You want to do something truly ridiculous and publish a book? A book that uses *multiple fonts*?

These are all pains. The pains will increase until you solve the underlying problems (Rule of System Administration #22). You overcome each pain by learning. The more you know, the more you become capable of overcoming pain. With enough practice, you can surmount all but the most exquisite insoluble agonies, and enduring those is excellent practice for the non-computing (aka "important") parts of your life.

I will, however, offer a few hints for your edification. If you're using a newish desktop or laptop, chances are you have at least 2GB of RAM and can thus use ZFS. If you care about the data on a single-disk system, give that important data its own datasets and set copies=2 on them. That will enable ZFS self-healing. Redundant disk setups like mirrors or RAID-Z already keep integrity data support self-healing. Use boot environments. If you're using a tiny

low-memory system, you're stuck with UFS. While UFS lacks the fancy features of ZFS, it has been knocked around like a stuffed bunny in a clothes dryer since 1983. All the sensible and most of the nonsensible bugs have been beaten out of it. You'll be fine.

Starting from the console and building up to a working desktop will teach you more than you thought possible. Specifically, it will teach you that you should have bought chickens.

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

MICHAEL W LUCAS is the author of *Absolute FreeBSD*, *Run Your Own Mail Server*, and *$ git commit murder*. The hip-mounted keyboard thing is completely real. Learn more at https://mwl.io.

Books that will help you. Or not.

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

— John Baldwin
FreeBSD Journal Editorial Board Chair

https://mwl.io

Contents

Consolations for Kernel Hackers

BY TOM JONES

You turn the machine on, it buzzes and whirrs, and there is a chime. The display jumps to a bright, but black screen, a first sign that the sounds aren't just distant movements. Some text from Phoenix or American scrolls by, and then you are shown the wonder of the FreeBSD boot prompt. 10 seconds later, the countdown and text begin scrolling. For some, this is a primitive machine, but for many of us, this is the best part. Being close to the metal at the start.

Kernel messages start to scroll by, beginning with the project copyright and continuing with devices as they are discovered and attached, until the kernel hands over to userspace. The text changes in aspect to indicate this, and we start seeing services start. All kernel hackers have sat watching this text scroll by, looking for the message they added to a new driver or an old piece of code that isn't behaving as we all thought it would. We are waiting for our `printf` to be called, proving that we have built and run a kernel with our changes, that we have finally tamed the build process, and can get to real work.

The machines we use now are quite different from those that UNIX was designed around.

All developers have used `printf` debugging and print statements to track execution flow, figure out how or if things are working, and where they connect. Most of the time, this core part of the kernel functions flawlessly; we get text output, or the kernel has died before it can be shown. This cinema is backed by a console driver, a piece of code that sits right at the bottom of `printf` at the interface between hardware and software. If you are embroiled in working on a console driver or in a very early boot, you want to know how the console works normally, so you can make it work abnormally and get some debug information you might not be able to otherwise.

Terminals and Consoles

The machines we use now are quite different from those that UNIX was designed around, which means that the boundary between the system console, an access terminal, and a graphical output isn't very clear. In the old days, the system would come up with a dedicated system console, something that was in or near the machine room where the computer lived. Terminals could live somewhere else quieter, either in a terminal room or on some lucky person's desk.

The console, though, was the heart of the machine; if the system needed to tell an operator about an exciting or dangerous failure, it would pop up on the console. The console is a core component of the system; most of the time, it is chatting away, printing messages on graphics output, which are unconnected or out to a serial port with no cable connected.

Most of the time, these messages go unused. They are the startup text the kernel outputs as it finds, discovers, and configures devices. While operating, these messages indicate hardware and system errors and notify us of buffer overflows and exhausted limits. When we need to see the messages, we can pull the buffer with the **dmesg** command. Usually, that is all anyone really needs.

But kernel hackers are an odd bunch, and sadly, most of the time, **printf** debugging remains the best way to track down errors and get signs of life from a new driver. If you get distracted and start digging into how exactly **printf** works in the kernel, you eventually start unearthing **cn*** functions and the console drivers' dispatch table.

Walking Down from printf

For most developers, kernel code is low-level, but our highest point today is any piece of code that can call **printf(9)**. Let's walk down from **printf** through some of the layers to get a sense of what's happening before a console driver is called. **printf(9)** lets us see what is usually the simplest case. **printf(9)** and its friends are implemented in **sys/kern/subr_prf.c**; the functions are mostly wrappers around **_vprintf**, which also handles outputting any buffered messages and putting bytes onto console devices via **kvprintf**.

```
static int
_vprintf(int level, int flags, const char *fmt, va_list ap)
{
    struct putchar_arg pca;
    int retval;
#ifdef PRINTF_BUFR_SIZE
    char bufr[PRINTF_BUFR_SIZE];
#endif
    TSENTER();
    pca.tty = NULL;
    pca.pri = level;
    pca.flags = flags;
#ifdef PRINTF_BUFR_SIZE
    pca.p_bufr = bufr;
    pca.p_next = pca.p_bufr;
    pca.n_bufr = sizeof(bufr);
    pca.remain = sizeof(bufr);
    *pca.p_next = '\0';
#else
    /* Don't buffer console output. */
    pca.p_bufr = NULL;
#endif
    retval = kvprintf(fmt, putchar, &pca, 10, ap);
    #ifdef PRINTF_BUFR_SIZE
    /* Write any buffered console/log output: */
    if (*pca.p_bufr != '\0')
          prf_putbuf(pca.p_bufr, flags, level);
#endif
    TSEXIT();
    return (retval);
}
```

It parses the format string, which resolves to a call to **putchar()**. **putchar** calls **cnput_char**, and finally we call into the driver dispatch table for driver-specific calls. **kvprintf** takes a function pointer to use for the actual outputting of bytes.

```
int
kvprintf(char const *fmt, void (*func)(int, void*), void *arg, int radix, va_list ap)
{
#define PCHAR(c) {int cc=(c); if (func) (*func)(cc,arg); else *d++ = cc; retval++; }
     char nbuf[MAXNBUF];
     char *d;
     const char *p, *percent, *q;
     u_char *up;
     int ch, n, sign;
     uintmax_t num;
```

In our example above, the function point is to **putchar**

```
/*
 * Print a character on console or users terminal. If destination is
 * the console then the last bunch of characters are saved in msgbuf for
 * inspection later.
 */
static void
putchar(int c, void *arg)
{
    struct putchar_arg *ap = (struct putchar_arg*) arg;
    struct tty *tp = ap->tty;
    int flags = ap->flags;

    /* Don't use the tty code after a panic or while in ddb. */
    if (kdb_active) {
          if (c != '\0')
                 cnputc(c);
          return;
    }
    if ((flags & TOTTY) && tp != NULL && !KERNEL_PANICKED())
           tty_putchar(tp, c);
    if ((flags & TOCONS) && cn_mute) {
           flags &= ~TOCONS;
           ap->flags = flags;
     }
     if ((flags & (TOCONS | TOLOG)) && c != '\0')
           putbuf(c, ap);
}
```

cnputc lives at the bottom of the kernel with many paths to it.

```
void
cnputc(int c)
{
    struct cn_device *cnd;
    struct consdev *cn;
```

```
        const char *cp;
        if (cn_mute || c == '\0')
                return;
    (2)
    #ifdef EARLY_PRINTF
        if (early_putc != NULL) {
                if (c == '\n')
                        early_putc('\r');
                early_putc(c);
                return;
        }
    #endif
      (1)    STAILQ_FOREACH(cnd, &cn_devlist, cnd_next) {
                cn = cnd->cnd_cn;
                if (!kdb_active || !(cn->cn_flags & CN_FLAG_NODEBUG)) {
                        if (c == '\n')
                                cn->cn_ops->cn_putc(cn, '\r');
                        cn->cn_ops->cn_putc(cn, c);
                }
        }
        if (console_pausing && c == '\n' && !kdb_active) {
                for (cp = console_pausestr; *cp != '\0'; cp++)
                        cnputc(*cp);
                cngrab();
                if (cngetc() == '.')
                        console_pausing = false;
                cnungrab();
                cnputc('\r');
                for (cp = console_pausestr; *cp != '\0'; cp++)
                        cnputc(' ');
                cnputc('\r');
        }
}
```

cnputc has two important pieces of code, the code in the loop at (1) walks a list of attached consoles and calls the consoles **cn_putc** callback. For reasons related to locking in and the debugger, it doesn't call these methods if the kernel debugger **kdb** is active.

This is where FreeBSD implements support for multiple consoles. If you ever wondered how, you get a glass terminal on a graphical display and a serial port, that's where it happens. The code at (2), wrapped in **#ifdef EARLY_PRINTF**, is incredibly platform-specific, enabling output from the first bytes of the kernel image. If defined and supported on your hardware, you can retrieve bytes from the loader handover.

As an example of how platform-specific **early_putc** is, here is the **x86/amd64** implementation.

```
#if CHECK_EARLY_PRINTF(ns8250)
#if !(defined(__amd64__) || defined(__i386__))
#error ns8250 early putc is x86 specific as it uses inb/outb
#endif
static void
```

```
uart_ns8250_early_putc(int c)
{
	u_int stat = UART_NS8250_EARLY_PORT + REG_LSR;
	u_int tx = UART_NS8250_EARLY_PORT + REG_DATA;
	int limit = 10000; /* 10ms is plenty of time */
	while ((inb(stat) & LSR_THRE) == 0 && --limit > 0)
		continue;
	outb(tx, c);
}
early_putc_t *early_putc = uart_ns8250_early_putc;
#endif /* EARLY_PRINTF */
```

This **early_putc** uses the x86 **outb** instruction to transmit bytes on a serial port, but it has never been given support for working with MMIO (memory mapped IO) UARTs which are now quite common.

(As I was writing this article, MMIO UART support on arm64 was added.)

Console Life Cycle

To look at the console life cycle, it is best to start from the FreeBSD Loader. Booting FreeBSD requires two Operating System components, the kernel and our boot loader (**loader(8)**). All hardware platforms start with some sort of firmware. In old x86 systems, this was the bios looking for a boot block. Modern x86 systems start via firmware, which presents the EFI interface. Non-PC systems boot in different ways. In embedded devices, a boot ROM on the system-on-a-chip tests and tries different boot media; the first one that works is used. On a device, that might mean that the boot ROM tries in order:

- NAND flash
- SD Card
- USB media
- A USB firmware mode

In many cases, the boot firmware will document what it is trying over that good old reliable serial interface. Typically, on ARM and RISC-V systems, a second-stage bootloader, called a secondary program loader, or SPL, is run. Almost always for us, this is u-boot. Early firmware will only print to a serial port, but u-boot in many cases has enough drivers together to be able to use a framebuffer device if possible.

Loader is modular and runs in many different forms depending on the base platform; we run from u-boot, as an EFI application, on top of open firmware, and in other places. For bhyve virtual machine images, the FreeBSD loader runs as a userspace application. Most of these platforms use a static console as the default output method and can, if possible, be configured to use additional outputs.

FreeBSD's loader runs as a chained application or a sub-application on top of the firmware or u-boot. U-boot will run EFI applications, making it much easier to create usable FreeBSD boot media. Without any other configuration, FreeBSD's loader determines what to use as the system console (the place where messages go). On an EFI system, we will use the EFI console, which has the unfortunate property of sometimes using buggy EFI interfaces to output characters. If you have ever used FreeBSD loader on a serial port on an EFI system, you might have seen duplicated bytes on the serial port — this is due to the EFI output routine being used in the usual EFI cons output and the EFI serial console output.

Loader can be configured to use multiple consoles via the config file or from the command line by adding more consoles to the **consoles** environment variable.

```
> set console="efi comconsole"
```

Loader passes configuration information to the kernel in the kernel image, along with some variables. The console type and whether multiple consoles are used are controlled by the **boothow** variable, which is set to an RB_* value. Historically (all the world's a VAX!), these were the system call arguments to **reboot(9)**, but they have been reused and become intertwined over time. From loader, we frequently get RB_SERIAL (use a serial port), RB_MULTIPLE (use multiple consoles), RB_SINGLE (boot to single-user), and RB_VERBOSE, which sets bootverbose in the kernel and fills up your message buffer.

Starting the Kernel

Eventually loader will start the selected kernel and pass along the arguments and environment. Everything that loader has done has been temporary, and all the work has to be done again in the kernel to get a working system that the kernel is in control of. Console initialisation is tied closely to platform-specific hardware. Each FreeBSD platform has its own machine-dependent code that runs and sets up memory and devices before the system can perform machine-independent work.

On most platforms this is an **init*** call (like **initarm**, **init386**, **initriscv** or the frustratingly unmatching **powerpc_init**), but on amd64 this is **hammer_time**. Each of these routines may try twice to initialise consoles (currently only i386 and amd64 do) based on configuration from loader. The relevant part of **hammer_time** looks like this:

```
	/*
	 * The console and kdb should be initialized even earlier than here,
	 * but some console drivers don't work until after getmemsize().
	 * Default to late console initialization to support these drivers.
	 * This loses mainly printf()s in getmemsize() and early debugging.
	 */
	TUNABLE_INT_FETCH("debug.late_console", &late_console);
	if (!late_console) {
		cninit();
		amd64_kdb_init();
	}

	getmemsize(physfree);
	init_param2(physmem);

	/* now running on new page tables, configured,and u/iom is accessible */

#ifdef DEV_PCI
	/* This call might adjust phys_avail[]. */
	pci_early_quirks();
#endif

	if (late_console)
		cninit();
```

You can see from the example that two attempts to call **cninit** are possible, depending on the value of the **late_console** variable, which defaults to true. Sandwiched between these two calls is the discovery and allocation of system memory. Before one of these calls to **cninit** it is not possible to output anything to the system console (it doesn't exist!), any **printf** before **cninit** will evaporate into the ether.

The exception is if **eary_putc** is defined as described, then you can do some early debugging. **cninit** walks a statically defined list of consoles and tries to find the best possible console.

```
	/*
	 * Find the first console with the highest priority.
	 */
	best_cn = NULL;
	SET_FOREACH(list, cons_set) {
		cn = *list;
		cnremove(cn);
		/* Skip cons_consdev. */
		if (cn->cn_ops == NULL)
			continue;
		cn->cn_ops->cn_probe(cn);
		if (cn->cn_pri == CN_DEAD)
			continue;
		if (best_cn == NULL || cn->cn_pri > best_cn->cn_pri)
			best_cn = cn;
		if (boothowto & RB_MULTIPLE) {
			/*
			 * Initialize console, and attach to it.
			 */
			cn->cn_ops->cn_init(cn);
			cnadd(cn);
		}
	}
	if (best_cn == NULL)
		return;
	if ((boothowto & RB_MULTIPLE) == 0) {
		best_cn->cn_ops->cn_init(best_cn);
		cnadd(best_cn);
	}
	if (boothowto & RB_PAUSE)
		console_pausing = true;
	/*
	 * Make the best console the preferred console.
	 */
	cnselect(best_cn);
#ifdef EARLY_PRINTF
	/*
	 * Release early console.
	 */
	early_putc = NULL;
#endif
```

Consoles are sorted by a priority set by the console driver when its **cn_probe** function is called. At the end, we must have a console (there is always a default), and we release **early_putc** so that it is gone forever. I didn't show the matching code in **loader.efi**, but it is very similar. The important exception for anyone hacking on EFI consoles is the lack of an **early_putc** mechanism to give you some last chance hope at practical **printf** debugging. However, this can be hacked around. In the EFI environment, the output routine is available, and if you just hang on to one of the consoles while doing the **init**, you can get more visibility into what is happening.

In the EFI environment, the output routine is available.

Available Consoles

The system consoles are defined at kernel build time. In **cninit** above, we walk through **cons_set** — a linker set generated for us by the build process. Each console is not declared as a normal device driver; instead, it is defined with the CONSOLE_DRIVER macro, which adds the console to the linker set. As of 16-CURRENT, linker sets aren't populated on kernel module load, meaning you cannot dynamically add console drivers to the kernel at run time.

The CONSOLE_DRIVER macro takes the name of the console to add to the linker set and handles the creation of all the callbacks based on the name:

```
CONSOLE_DRIVER lives in sys/sys/cons.h
#define CONSOLE_DEVICE(name, ops, arg)                          \
        static struct consdev name = {                     \
                .cn_ops = &ops,                    \
                .cn_arg = (arg),                  \
        };                                  \
        DATA_SET(cons_set, name)
#define CONSOLE_DRIVER(name, ...)                            \
        static const struct consdev_ops name##_consdev_ops = {          \
                /* Mandatory methods. */              \
                .cn_probe = name##_cnprobe,              \
                .cn_init = name##_cninit,             \
                .cn_term = name##_cnterm,                \
                .cn_getc = name##_cngetc,                \
                .cn_putc = name##_cnputc,                \
                .cn_grab = name##_cngrab,                \
                .cn_ungrab = name##_cnungrab,              \
                /* Optional fields. */      \
                __VA_ARGS__                   \
        };                              \
        CONSOLE_DEVICE(name##_consdev, name##_consdev_ops, NULL)
```

The macro defines the mandatory callbacks that a console driver must implement and provides space to add extra code. Currently, **uart_tty** takes advantage of this to add a resume method. The console callbacks are all quite minimal; we really want the console to work after all. **cn_probe** needs to set the console priority (**cn_pri**) to something other than CN_DEAD for the console driver to be considered by **cninit**.

`cn_init` is called for selected console devices by `cninit` (either the best one or all non-dead consoles if we are booting with `RB_MULTIPLE`). In `cn_init` the console driver can establish state, as an example `uart_cninit` stores a cookie so the console isn't initialised twice. `cn_term` is called if the console driver is ever removed to tidy things up. `cn_grab` and `cn_ungrab` are used to disable and enable interrupts if the kernel is going to print a series of characters (or read one in). Finally, `cn_putc` and `cn_get` handle output and input for the console device. The magic we care about.

Outputting Bytes

There are many different devices that can be used as a system console. It is sometimes hard to separate the pc from the x86 (or AMD64) platform, but a big part of what makes it a platform rather than just a processor architecture is the ecosystem of peripherals and hardware that can be expected to be present. On the PC we can expect to see 1 or more UARTs (serial ports). This is so fundamental that FreeBSD as of 15.0 still assumes there will be two and statically configures them. UARTs were commonly hooked out to explicit instructions (`inb` and `outb`), which allowed a caller to send data out the serial port with a single instruction.

This simple interface is deliberate; if you are bringing up a machine from nothing, it is very handy to be able to get debug information from the system with a minimal proof of concept.

This simple interface allows a program (or a driver) to interact with the outside world. If you are the operator of a new machine and want to know what the system is up to, then connecting a serial printer, a teletype, or, in the year 2026, a terminal program and serial cable let you get debug information from even the smallest code examples.

Instructions for input and output of bytes on a serial port put limitations on processor and hardware designers. You can't expect an instruction for every device, and so we have moved mostly to memory-mapped peripherals. The same 8250 and 16650 devices used in the original PC are available in modern systems, with compatible hardware blocks in modern system-on-chip designs. Rather than using `inb` and `outb`, they are memory-mapped peripherals.

There are many different devices that can be used as a system console.

Other systems expose slightly different devices for their console, but on almost every platform, this boils down to using the old National Semiconductor 8250 or 16650 UART interface for getting bytes onto the wire. This means we see platform-specific console devices, but many, many of them resolve down to the same battle-tested code.

I don't think anyone who has used a computer in the last 3 decades would be very happy with just a serial port. From the beginning, the pc supported graphical output, and the BIOS made accessing it practical with very simple early program loaders. With a serial port and a graphical output, it doesn't seem right to explicitly bake in `outb` into `printf` for getting bytes out of a serial port. So, while a very simple console driver is possible, we end up building a console driver subsystem that lets us reuse much of the code while still allowing platform and hardware-specific variations.

Loader

So far, we have mentioned loader quite a few times as it was relevant during kernel start-up, but it is an operating system in its own right. Loader prepares the kernel's operating environment and configures the root volume, which provides the userspace environment for kernel modules. All platforms that FreeBSD runs on have a loader of some sort (well, we can directly boot the kernel in some cases, but that leaves much functionality on the table).

Loader is started either by the platform firmware or a secondary loader. In most deployments, the loader runs in the EFI environment (on arm64, arm32, amd64, and riscv). This environment provides a lot of early services to loader, making its job easier and portability more straightforward. As loader runs in many environments and supports 2 scripting environments (Lua and Forth, yes, Forth in 2026), it provides some of its own abstractions to make core services easier to use. Loader also has console drivers, and they look very similar to the kernel ones.

```
stand/common/bootstrap.h:
  /*
   * Modular console support.
   */
  struct console
  {
       const char      *c_name;
       const char      *c_desc;
       int             c_flags;
  #define C_PRESENTIN     (1<<0)      /* console can provide input */
  #define C_PRESENTOUT    (1<<1)      /* console can provide output */
  #define C_ACTIVEIN      (1<<2)      /* user wants input from console */
  #define C_ACTIVEOUT     (1<<3)      /* user wants output to console */
  #define C_WIDEOUT       (1<<4)      /* c_out routine groks wide chars */
       /* set c_flags to match hardware */
       void    (* c_probe)(struct console *cp);
       /* reinit XXX may need more args */
       int             (* c_init)(int arg);
       /* emit c */
       void            (* c_out)(int c);
       /* wait for and return input */
       int             (* c_in)(void);
       /* return nonzero if input waiting */
       int             (* c_ready)(void);
  };
```

Loader console drivers have in and out methods, as the kernel ones do; an init method, a probe method, and a ready method to speed up polling for input. Rather than priorities, loader console drivers have a flags argument. Each variant of loader has its own **conf.c** file, which defines its platform-specific features and implementations. One of these features is a list of consoles that can be used on that platform. For **loader.efi** we have:

```
#if defined(__amd64__) || defined(__i386__)
extern struct console comconsole;
extern struct console nullconsole;
```

```
extern struct console spinconsole;
#endif
struct console *consoles[] = {
     &efi_console,
     &eficom,
#if defined(__aarch64__) && __FreeBSD_version < 1500000
     &comconsole,
#endif
#if defined(__amd64__) || defined(__i386__)
     &comconsole,
     &nullconsole,
     &spinconsole,
#endif
     NULL
};
```

For an amd64 machine our full consoles list is:

- **`efi_console`**: Console using the efi output routines
- **`eficom`**: Console using the efi serial output routines. This one can be a little strange and lead to duplicated output as the EFI implementation uses the framebuffer and the serial port.
- **`comconsole`**: Console on a traditional COM port, a serial console using the
- **`nullconsole`**: All output bytes disappear into the void.
- **`spinconsole`**: Each output byte progresses a spinner by one step; this gives a visual indication of something happening, but no way to debug or provide input.

There are platform-specific consoles for the userboot loader (this is for start bhyve instances), the kboot (boot from Linux), u-boot, and other platforms. On amd64 **`comconsole`** and **`eficom`** map to the same driver functions.

EFI Loader Console init

The load **`cons_probe`** function handles discovering consoles and configuring them. It is called very early in **`efi/loader/main.c`**:

```
EFI_STATUS
main(int argc, CHAR16 *argv[])
{
...
#if !defined(__arm__)
      efi_smbios_detect();
#endif

      /* Get our loaded image protocol interface structure. */
      (void) OpenProtocolByHandle(IH, &imgid, (void **)&boot_img);

      /* Report the RSDP early. */
      acpi_detect();

      /*
       * Chicken-and-egg problem; we want to have console output early, but
       * some console attributes may depend on reading from eg. the boot
```

```
         * device, which we can't do yet.  We can use printf() etc. once this is
         * done. So, we set it to the efi console, then call console init. This
         * gets us printf early, but also primes the pump for all future console
         * changes to take effect, regardless of where they come from.
         */
        setenv("console", "efi", 1);
        uhowto = parse_uefi_con_out();

#if defined(__riscv)
        /*
         * This workaround likely is papering over a real issue
         */
        if ((uhowto & RB_SERIAL) != 0)
              setenv("console", "comconsole", 1);
#endif
        cons_probe();
        /* Set print_delay variable to have hooks in place. */
        env_setenv("print_delay", EV_VOLATILE, "", setprint_delay, env_nounset);

        /* Set up currdev variable to have hooks in place. */
        env_setenv("currdev", EV_VOLATILE, "", gen_setcurrdev, env_nounset);

        /* Init the time source */
        efi_time_init();
```

The chicken-and-egg comment is common in the tree. Basically, we need to be able to debug what is happening as early as possible, and a console is the easiest way to set that up. The EFI environment we have right now can output strings, but we don't have console infrastructure. We play a trick on all platforms and statically pick a console before looking for consoles.

Console probing happens before we can read from storage, so it starts with assumptions and then reassesses. On EFI we can output data before `cons_probes` by making a direct EFI with `ST->ConOut->OutputString`. We have console infrastructure, and that lets us use `printf`, so we should. `cons_probe` walks the list and finds a working console. The selected console is added to the `console` environment variable in loader and can be changed or added to by environment variables from `loader.conf` or on the command line.

Once consoles are selected, they are plummed into `printf` in a similar way to kernel consoles.

```
common/console.c:
  void
  putchar(int c)
  {
        int     cons;
        /* Expand newlines */
        if (c == '\n')
               putchar('\r');
        for (cons = 0; consoles[cons] != NULL; cons++) {
               if ((consoles[cons]->c_flags & (C_PRESENTOUT | C_ACTIVEOUT)) ==
```

```
                    (C_PRESENTOUT | C_ACTIVEOUT))
                      consoles[cons]->c_out(c);
        }
        /* Pause after printing newline character if a print delay is set */
        if (print_delay_usec != 0 && c == '\n')
        delay(print_delay_usec);
}
```

Debugging a Console Driver

So, there is all the magic in a console driver, and on reflection, it is a bit of a dim light when we look at it in detail. `printf` is quite simple, and it is driving a single character output routine. It is similar between the kernel and loader, I know you are wondering why I repeated the dive with loader, but this was to show that it is the same and that debugging tricks that work in the kernel will probably work in loader environment. If you are stuck pre-console or have an issue that perturbs the working console, you will reach for anything you can to debug what is happening.

So how can we debug issues with console drivers?

The first and best two suggestions I have are to use a debugger or a second console. If you can use a debugger for loader or the early boot environment, then you really should; it will cut through all the rest of the stuff. You may be able to use a debugger if you have a second working console (the kernel debugger leverages the console infrastructure) or if you can control the machine under development holistically. Either using an emulator such as `qemu` or with a hardware debug interface like JTAG. Setting these up is beyond the scope of this article, and I know that if you've read this far, you are thinking, "But I'm only doing this thing because I can't debug!"

A second console is a great option for when you need to debug the console, but it probably won't help so much early in the kernel. If you have a working console and questions about the non-working console, you can bypass much of the infrastructure and either configure things you know will be there statically or directly hook into the console functions you want to use.

The first and best two suggestions I have are to use a debugger or a second console.

This is where the linker set for the list of consoles and the static list in loader really helps. Just pull out the console you want from the list and take `put_c` and do what you need. Doing this skips all the locks and loses you `printf`, but it also skips all the locks, and it might let you get debugging information out.

I have used this direct capture to debug a loader console driver; loader doesn't have an `early_putc`, so hacks were needed. Usually, during `cons_probe` in loader, we walk all the consoles and remove the `ACTIVE_OUT` flag; this means that if we use `printf`, we get no data out. To debug console detection, I statically configured the first console (knowing it would be there and working) and used `printf` to debug the start-up of my driver of interest.

The change looked something like this:

```
diff --git a/stand/common/console.c b/stand/common/console.c
index 65ab7ffad622..97201a7479b3 100644
--- a/stand/common/console.c
+++ b/stand/common/console.c
@@ -107,14 +107,18 @@ cons_probe(void)
   env_setenv("twiddle_divisor", EV_VOLATILE, "16", twiddle_set,
     env_nounset);

+   /* capture the first console and keep it working while parsing others */
+    consoles[0]->c_flags |= C_ACTIVEIN | C_ACTIVEOUT;
+    consoles[0]->c_init(0);
+
    /* Do all console probes */
-    for (cons = 0; consoles[cons] != NULL; cons++) {
+    for (cons = 1; consoles[cons] != NULL; cons++) {
          consoles[cons]->c_flags = 0;
          consoles[cons]->c_probe(consoles[cons]);
     }
     /* Now find the first working one */
     active = -1;
-    for (cons = 0; consoles[cons] != NULL && active == -1; cons++) {
+    for (cons = 1; consoles[cons] != NULL && active == -1; cons++) {
          consoles[cons]->c_flags = 0;
          consoles[cons]->c_probe(consoles[cons]);
          if (consoles[cons]->c_flags == (C_PRESENTIN | C_PRESENTOUT))
```

The final hack I have with console drivers is to explicitly take control of the output function. **early_putc** is a good feature, but it only works for some devices and is dropped at **cons_probe**. If you pull out the **cn_putc** for a console driver you need to use or test, then you have an output method you can use sans **printf** when you need it.

Recently, trying to understand why a uart wouldn't output anything I did, added this crime to **cons_probe**:

```
  if (uart == NULL || cp == NULL) {
        printf("uart is NULL\n");
  } else {
      // this won't work if we boot multi, it'll panic on reinit
        printf("trying to init\n");
        uart->cn_ops->cn_init(uart);
        printf("trying to print\n");
        uart->cn_ops->cn_putc(uart, 'H');
        uart->cn_ops->cn_putc(uart, 'E');
        uart->cn_ops->cn_putc(uart, 'L');
        uart->cn_ops->cn_putc(uart, 'L');
        uart->cn_ops->cn_putc(uart, '\r');
        uart->cn_ops->cn_putc(uart, '\n');

#define TEST_COUNT 100
        printf("Tring to print %d 4's\n", TEST_COUNT);
        for (int i = 0; i < TEST_COUNT; i++)
```

```
            uart->cn_ops->cn_putc(uart, '4');

        uart->cn_ops->cn_putc(uart, '\r');
        uart->cn_ops->cn_putc(uart, '\n'ß);
}

for( int i = 0; i < 40; i++) {
        DELAY(100000);
}
```

We manually set up the uart and output bytes, I was trying to understand why no data was coming out of the port, and that loop outputting `4` gave me something to sync an oscilloscope to.

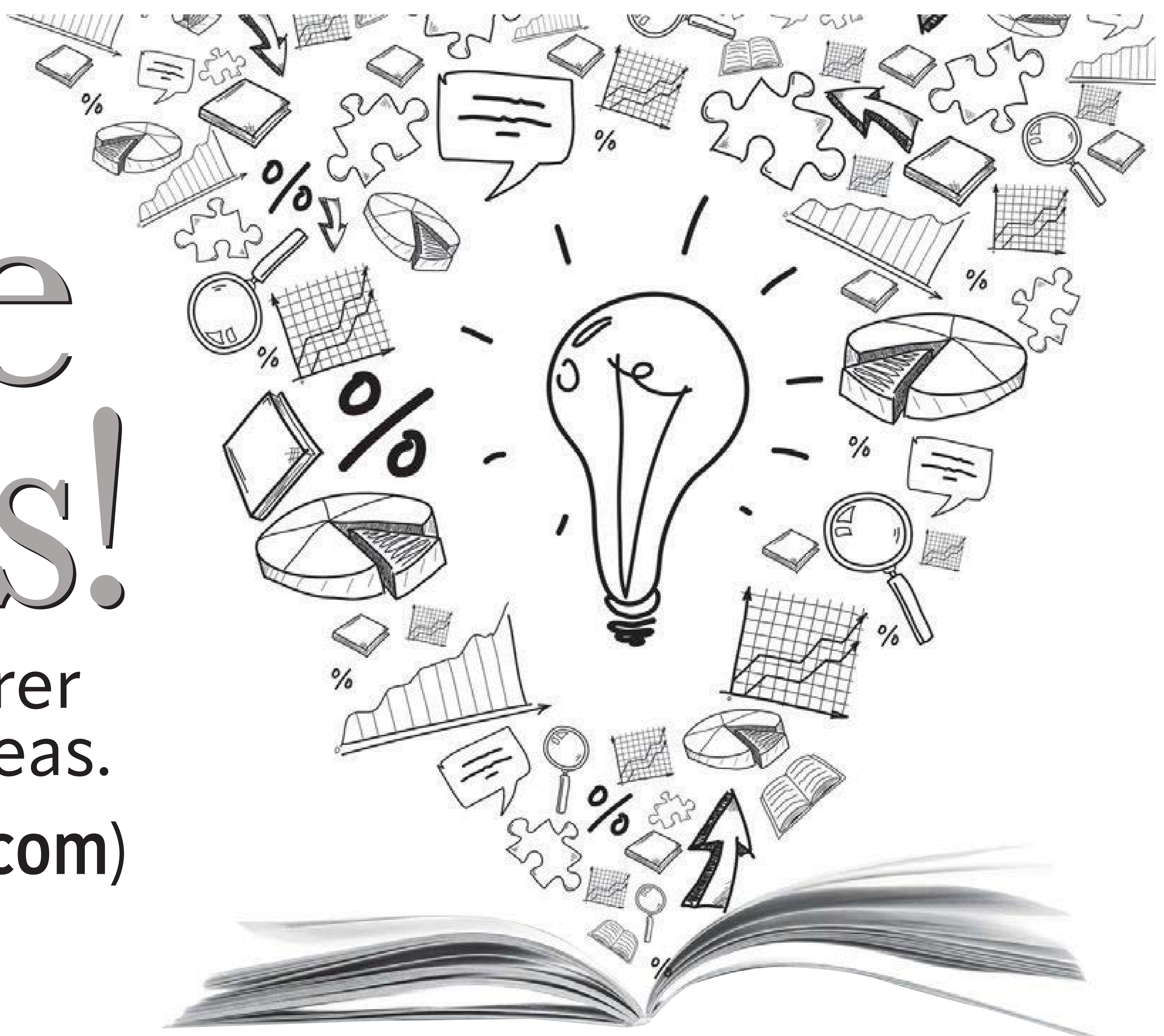
Consolations

If you have to debug a console driver or work in loader or the kernel in the pre, or very early boot, then you will take what you can get, to get data out of the system. We don't have to toggle a GPIO; thankfully, the console system is pretty simple in design and use. That means it should work, but if it doesn't, you can bend it to output data when you need it to. These sorts of intermediate hacks are horrible and never appear in the source tree, but they can be very useful when you need to debug something, and it won't comply.

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

Write For Us!

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



Let Sleeping CPUs Lie — S0ix

BY AYMERIC WIBO

Modern laptops promise a kind of magic. Shut the lid or press the sleep button, toss it in a backpack, and hours, days, or weeks later, it should wake up as if nothing happened with little to no battery drain. This sounds like a fairly trivial operation — y'know, you're literally just asking for the computer to do nothing — but in that quiet moment when the fans whir down, the screen turns dark, and your reflection stares back at you, your computer and all its little components are actually hard at work doing their bedtime routine.

History of Power Management, or: The Long Road to Doing Nothing Efficiently

Let's take a stroll through the history of power management on computers to better understand where we came from.

The first "mobile" computers were more like semi-portable machines that, if they had batteries at all, weren't really expected to stay unplugged from AC for very long. With the Osborne 1, while it did have an aftermarket battery expansion pack, it was expected that you'd be powering it up and down as you ran between outlets.

The first true mobile computer, as we know it, was probably the Dulmont Magnum, released in 1983. It introduced "suspend-to-RAM" functionality for the first time, which in its case meant that components could be switched off while memory was retained in battery-backed CMOS RAM. A few other DOS machines shipped proprietary suspend solutions, but for the most part, they were fairly ad hoc.

A few other DOS machines shipped proprietary suspend solutions, but for the most part, they were fairly ad hoc.

In 1985, Intel released the i386 chips, the SL variant of which (itself released in 1990) targeted power-efficient machines such as laptops. This chip introduced SMM (System Management Mode), which allowed CPU execution to switch from the operating system to a region of memory written to and locked by the BIOS (the SMI handler, starting at SMBASE), which would run with higher privileges than ring 0 to run power management code.

This switch-over in execution was initiated by an SMI (System Management Interrupt), which could be asserted either by hardware (e.g., a power button press) or by the operating system via I/O access to a specific port. This method of power management had major flaws, not least that the operating system was a passive actor in this arrangement and completely at the mercy of the firmware. Indeed, if the SMI were asserted by hardware, the OS

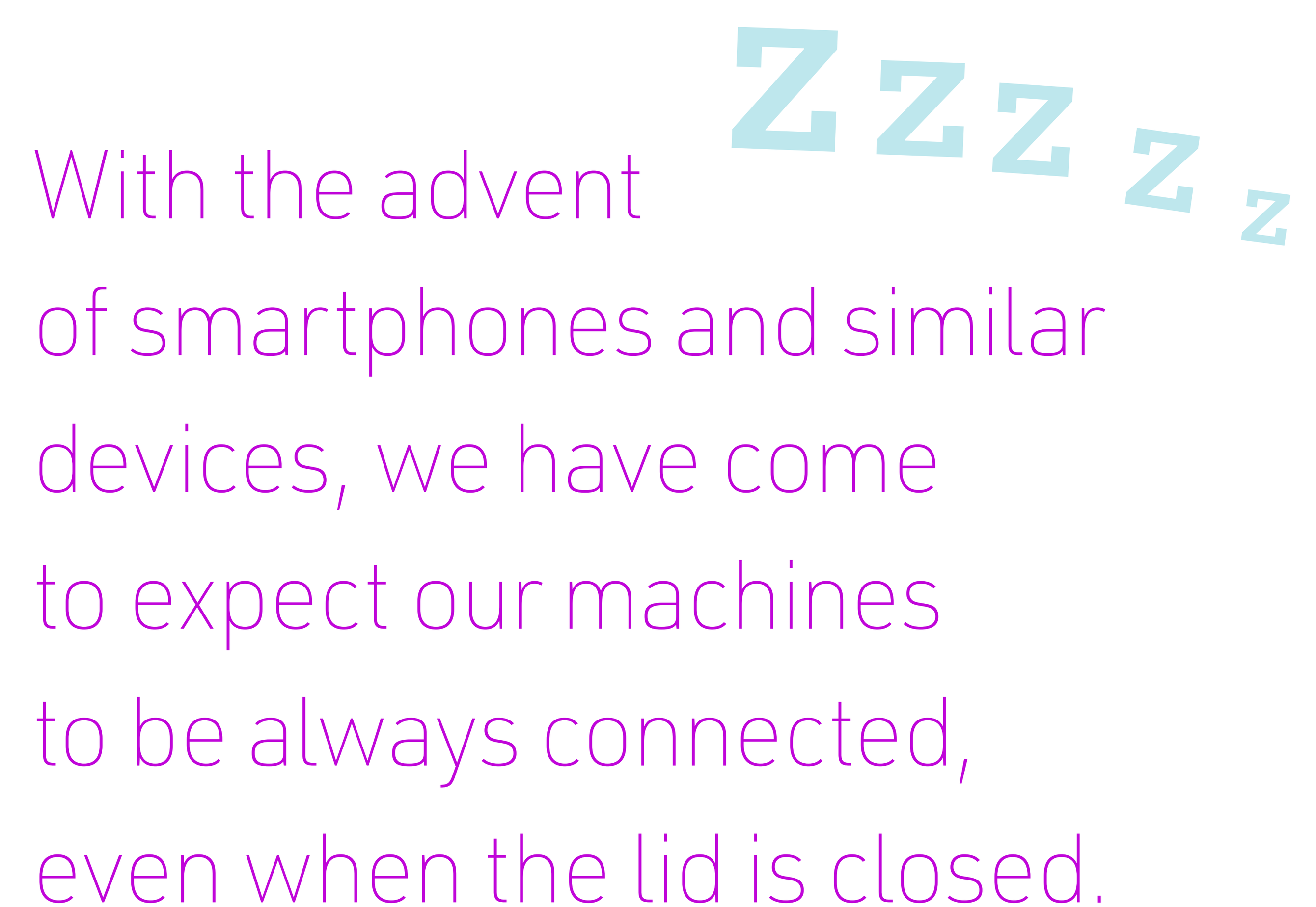
would be entirely sidestepped, and execution would be essentially flash-frozen, and it would thus be fully unable to prepare for suspend entry.

SMI handlers were also a very fruitful source of exploits — notably many of the "implants" from the NSA's infamous ANT catalogue — which remotely flashed BIOSes, allowing a malicious payload to be loaded at SMBASE, which would then run periodically during the system's operation.

Introduced in 1992, APM (Advanced Power Management) finally provided a standardized power management interface for x86 platforms. Instead of the OS being interrupted at the hardware's whim, when an event that would've previously triggered an SMI occurs, the firmware simply notifies the OS via a power management event. The OS then decides what to do in response, if anything, and can take all the time it needs to prepare before actually requesting a power transition from the firmware via an APM function (usually via BIOS interrupt 0x15).

This was an absolutely massive improvement over SMM for power management. One issue still persisted, though: APM still assumed BIOS vendors would write software not riddled with bugs, and these APM functions were asking a lot of these BIOSes.

APM was eventually superseded by ACPI (Advanced Configuration and Power Interface) in 1996, which was a much more comprehensive standard and interface for hardware and firmware to communicate with the operating system and provide their functionality. It is still used on modern machines today.



ACPI aimed to lug even more power management responsibilities from the firmware to the OS side of things, which it named "OSPM" (OS-directed Power Management). It defines multiple "S"-states, which are high-level power states the system could be in.

Most notable are S0 (the computer is fully awake and running), S3 (the computer is suspended-to-RAM), S4 (the computer is suspended-to-disk, or hibernated), and S5 (the computer is shut down). Fundamentally, however, suspending through S3 follows a similar approach to APM, since the "OS is informed of a power event, then deciding and preparing for suspend, and finally asking firmware to actually suspend" flow is maintained.

But as our hardware progresses and we come to expect more from it as the spoiled consumers we are, this flow, and actually the whole "the system is either fully on or suspended" paradigm, is starting to show its limitations.

With the advent of smartphones and similar devices, we have come to expect our machines to be always connected, even when the lid is closed. We want to receive notifications about new uploads from our favourite AI slop content creators, emails from that 500th newsletter we're subscribed to, and to wake the machine to notify us of calendar events.

To do this, we need to resume every so often or upon receiving a wake-on-LAN packet, just to, e.g., look at it a little more deeply and decide whether we actually want to wake the system for real or snooze and go back to sleep immediately. And this is a use case S3 is just too inflexible to properly accommodate — S3 is extremely costly in terms of energy consumed and time taken for firmware to start everything back up again.

S3 is all-or-nothing; if we want to wake from it, we pretty much have to act as if we were fully resuming each time.

If only there were a solution...

Enter... S0ix

Somewhere around 2018, in the everlasting quest to make computers do absolutely nothing as efficiently as possible and better and faster than ever before, Intel introduced S0ix. S0ix is actually a collection of states (S0i1, S0i2, S0i3, potentially &c), but which crucially all keep the system in the S0 — or awake — ACPI state. The bigger the "x", the deeper the sleep and the more the power saving, with the goal generally being S0i3. S0i0 is sometimes referred to as equivalent to S0, but I think that's somewhat confusing, so I won't use that term.

Contrary to S3 or APM in the past, the OS never actually explicitly asks the firmware to enter S0ix (erm, well, actually in practice it kind of does, but we'll get to that later). Instead, it just creates the conditions necessary for entry into an S0ix state and then lets the firmware decide when to transition the CPU to a given S0ix state on its own.

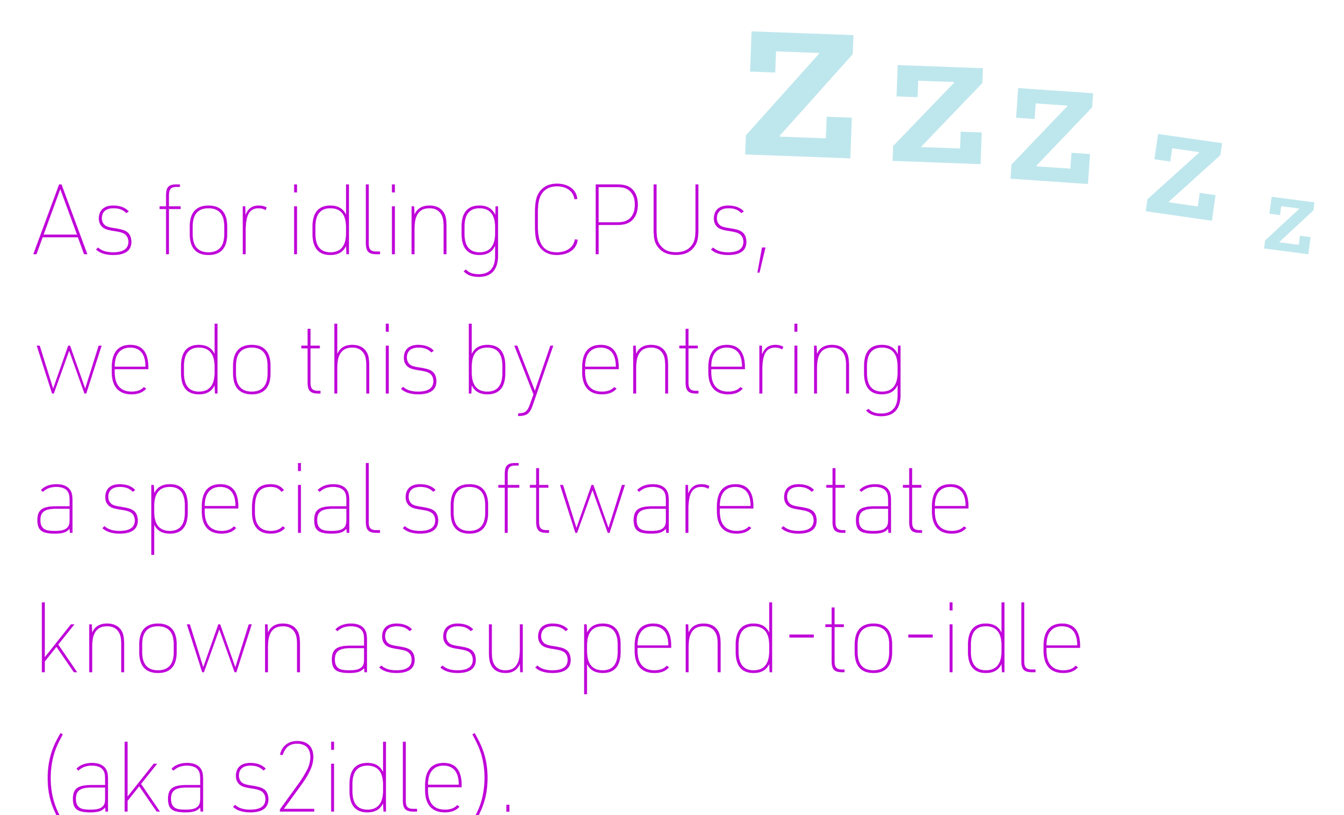
Those necessary conditions are:

- The devices (i.e., various components such as the GPU or USB controllers) on the machine are sufficiently powered down for a given S0ix state.
- The CPUs are all idling (that is, doing nothing) and have entered a low-power idle state.

Let's take a look at how we power down devices.

S0ix-enabled machines expose an ACPI device known as the SPMC (System Power Management Controller, sometimes also known as the PEP). Functions known as DSMs (Device-Specific Methods) in ACPI-land may be called on this SPMC device to provide hints to the platform about the OS's intention to suspend, but can also be used to get constraints about what the minimum power state a device needs to be in for firmware to want to enter an S0ix state.

Device power states are known as "D-states", and they range from D0 (fully on) to D3hot (off but powered) and D3cold (off but unpowered). We'll generally want to transition a device into D3cold anyway even if its D-state constraint doesn't require it to be if doing a full suspend, but we might want to keep a device in a more wakeful D-state to serve as a wake device — of course, if we power off a device meant to signal the system we would like to wake up, we would never be able to wake up.



So, to make things simple, these constraints from the SPMC just tell us whether keeping a device on during suspend would impede S0ix entry. As for idling CPUs, we do this by entering a special software state known as suspend-to-idle (aka s2idle), which pretty much takes the place S3 used to. Essentially, we just do this by stopping the scheduler clocks and pausing whatever the OS was executing to put all the CPUs in the system into a low-power idle state. These low-power idle states are known as C-states, and their entry methods, along with other information, are obtained either from the older `_CST` ACPI object or from its more modern replacement, the `_LPI` object.

Exiting the idle state is done by interrupting the CPU, so before idling, we disable all CPU interrupts except those we actually want to use to wake the CPU. These wake interrupts are usually just the SCI (ACPI System Control Interrupt), which let us know of GPEs (General Purpose Events), which could be something like pressing the power button or opening the lid (coarsely discriminated by the "GPE number").

SCIs can trigger for a variety of non-wake reasons, so we also configure ACPI as best we can to only trigger SCIs on actual wake GPEs, which is sometimes easier said than done. But we'll get to that later.

In theory, if both conditions are met, firmware can cut power to the CPU package and enter a S0ix state. In practice, though, things are a little more complicated and hardware-specific.

On AMD systems, for instance, the power management firmware (which they call PMFW) runs on a small on-die LatticeMico32 core known as the SMU (System Management Unit), which you'll also see more recently referred to as the "MP1" chip. This is ultimately responsible for clock and power gating the CPU package, and thus for transitioning the package in and out of S0ix states.

When the OS actually wants to suspend, it has to send the PMFW a little command to hint at it to enter S0i3. The SMU has its own requirements that must also be met, in addition to the regular S0ix requirements. The most notable of these requirements is for the GPU and USB4 controllers: recent versions of the amdgpu driver use different paths when entering S3 or S0ix, and USB4 controllers need to be explicitly powered off by an NHI driver. This is necessary even if the operating system doesn't support USB4, because the controllers are activated by the pre-OS connection manager (required for USB4 to work before the OS boots) and remain active by the time the OS boots.

In theory, if both conditions are met, the firmware can cut power to the CPU package and enter a S0ix state.

The SMU also communicates debugging information, letting the user know if the system entered S0i3 successfully and, if not, what prevented it from doing so. This information is exposed through the `dev.amdsmu.0.ip_blocks` and `dev.amdsmu.0.metrics` sysctl trees on FreeBSD.

Since the PMFW is running purely on-die, to let the wider platform know of our will to enter S0ix, we have to tell the SPMC we want to enter suspend through those "display off" and "entry" notification DSMs alluded to earlier. This usually prompts the platform to provide visual feedback to the user that the system is suspended, such as slowly fading the power button LED in and out on Framework laptops. It can also do some more subtle stuff, such as reducing the frequency of some GPEs that would otherwise constantly wake the CPU from idle and thus S0i3.

Indeed, on the Framework laptops, the EC (Embedded Controller) sends a GPE for the battery device about once per second in normal operation, and we cannot suppress this before entering suspend, as the battery GPEs share a number with the lid GPE. So if we wanted to disable those noisy battery GPEs, we'd have to sacrifice being able to wake the system with the lid, which is no good.

Hinting the SPMC of entry to suspend tells the EC to send a battery GPE only about once a minute, which is perfectly fine considering how quickly we can exit S0i3 and re-enter. And having our system wake every so often can actually be fairly convenient for tracking what the power draw and battery state look like over time while we're in suspend-to-idle (even though we could probably replicate this with the RTC alarm, but I digress).

Since the CPU might be woken from idle for a whole host of reasons that don't necessarily mean we want to wake the whole system, we put the CPU in an "s2idle loop" when it's idling. This allows us to actually look at the GPE we got from the system and decide whether it's worth waking for. This GPE could be as simple as noisy battery events or as complex as a wake-on-LAN event from the NIC, where, as mentioned earlier, we might want to examine a packet in a bit more detail before deciding whether to wake the system.

All in all, the full flow for entering suspend on S0ix enabled systems is the following:

- Suspend all userspace execution and mounted filesystems.
- Go through the device hierarchy and suspend all devices, usually by setting them to one of the D3 states and at least setting them to the minimum required D-state for S0ix as reported by the SPMC.
- Halt the scheduler clocks on all CPUs and disable all interrupts, except for SCIs.
- Start the "s2idle loop" and idle the CPUs.

Resuming is pretty much the first three steps, but in reverse.

Zzzzz

Why should I bother with this newfangled S0ix stuff when I can stick to S3?

Now, dear reader, at this point you might be thinking to yourself "well gee wizz, that's good and all but I don't need my laptop to respond to notifications while its sleeping. Why should I bother with this newfangled S0ix stuff when I can stick to S3?", and to that the unfortunate response is that most modern systems nowadays don't come with S3 support at all anymore, having completely replaced it with S0ix.

Supporting both is very complex for vendors, so most don't bother, especially considering most major operating systems support S0ix just fine now. Luckily, FreeBSD is well on its way to adding support for suspend-to-idle and S0ix.

You can see if your system supports suspend-to-idle by checking the `kern.power.supported_stype` sysctl, and you can configure ACPI to enter suspend-to-idle on certain events through the `hw.acpi` sysctl tree. Then, if your system supports it and FreeBSD supports it, an S0ix state should be entered automatically.

Currently, there is no unified way of checking this, but if you're on an AMD system with an SMU chip, you can use the aforementioned SMU sysctls.

Hibernate and ACPI S4

After all this work getting S0i3 to work, you might be a little disappointed to learn that in S0i3 (and S3, too, for that matter), you're not even getting very good power savings. There's a lot of variance between systems in this regard, but a system lasting only a few days in S0i3 is not out of the ordinary. That's where hibernate, or suspend-to-disk, comes in.

When suspending to memory, as in S3 or S0i3, your machine still has to constantly refresh DRAM, which isn't an inconsequential source of power drain, and this actually in-

creases as you add more memory to your system. With suspend-to-disk, you are copying memory to a hard drive and effectively completely turning off your computer — including DRAM — as if you were shutting it down (well, for all intents and purposes you *are* shutting it down). This means your machine is consuming almost no power at all. The only components actively consuming power are things like your EC if your machine has one, but that's just the same as if your machine were powered off anyway.

In ACPI S-state parlance, hibernate is S4.

At the very beginning, when dinosaurs still roamed the Earth, BIOS vendors implemented this in firmware via something called S4BIOS to ease adoption. In this arrangement, the BIOS was entirely responsible for copying OS memory to disk, much as suspend-to-RAM worked pre-APM.

Though FreeBSD supports S4BIOS, it hasn't been relevant for a long time, as most operating systems nowadays only support regular S4 hibernate. In S4, the operating system does most of the work of copying its memory to disk, while the firmware essentially just powers off the machine. In fact, you can totally implement hibernate with no firmware S4 support at all, just using your platform's poweroff functionality (which means S5 on ACPI platforms, or whatever else on non-ACPI platforms).

With suspend-to-disk, you are copying memory to a hard drive and effectively completely turning off your computer.

The main drawback of hibernate is the latency at entry and exit. You have to wait for the operating system to copy all pages to disk, then, on resume, go through much of the boot process and finally restore those pages from disk to memory. But we can actually combine the advantages of S0ix and hibernate into a hybrid approach!

By setting an RTC alarm, we can let the system stay in S0i3 for, say, an hour, and then wake up again and enter hibernate if the user has not woken the system up in the meantime. This means that if we wake up less than an hour before suspending, we get the super-fast resume times from S0ix, but if we leave the system suspended overnight or for multiple days, we get the much *much* improved battery life of hibernate. That does mean it will take significantly longer to resume the next time we wake the machine if we didn't wake it during the hour, but that's a pretty good compromise, all things considered.

We can also do other cool stuff with this hybrid approach, like configuring the ACPI `_BLT` (Battery Level Threshold) control method to wake the machine once the battery has reached a certain threshold — say 5% — so that we can enter hibernate and prevent a hard poweroff if the battery runs out while we are in S0ix.

State of Things in FreeBSD

The FreeBSD Foundation, as part of its "Laptop Support and Usability Project" effort, has been spearheading sponsored work on S0ix and hibernate. Much of the suspend-to-idle and S0ix code has already landed, and hibernate has recently been kicked off, but entry into S0i3 and especially resumption from it are not entirely reliable yet, and the necessary parts of the USB4 driver to make the SMU happy have not yet landed.

Furthermore, much of the focus has been on AMD systems and not so much Intel (though work on an initial PMC driver — their SMU equivalent — has recently begun). Even among AMD platforms, the only testing has so far occurred on Phoenix, with the `amdsmu` driver theoretically only supporting Cezanne, Rembrandt, and Phoenix chips.

The only USB4 controller to have been tested with the work-in-progress USB4 driver (`thunderbolt`) is Pink Sardine (found on Phoenix systems), though any generic quirkless USB4 controllers should also be supported.

Much of the remaining work at the time of writing focuses on reliably entering the deepest C-state (C3 on Phoenix systems), as CPUs often wake unexpectedly, which in turn causes the system to exit S0i3. But fingers crossed, FreeBSD will soon have support for modern standby and hibernate, and be fully caught up with the likes of Linux and Windows!

For a more technical and focused look at S0ix and its implementation in FreeBSD, plus some additional references, you can take a look at my blog: https://obiw.ac/s0ix

AYMERIC WIBO is a software freelancer from Belgium specializing in kernel development and graphics programming. He has been involved with FreeBSD since high school and is currently working for the FreeBSD Foundation on laptop power management and at Klara Inc.

The FreeBSD Project is looking for

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

Find out more by

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

Downloading the Software
freebsd.org/where.html

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

Already involved?

Don't forget to check out the latest grant opportunities at **freebsdfoundation.org**

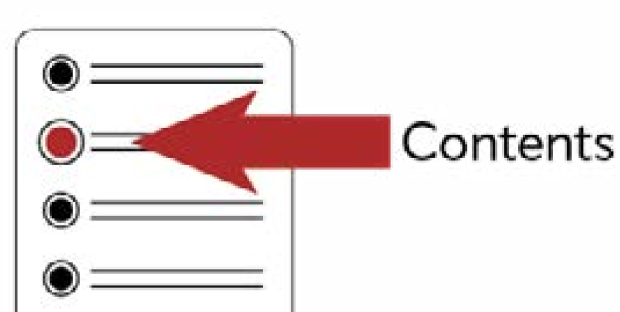
Help Create the Future. Join the FreeBSD Project!

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

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

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

The FreeBSD Community is proudly supported by



How the Foundation's Laptop Support & Usability Project Came Together

BY DEB GOODKIN

For years, we kept hearing the same thing from users: yes, FreeBSD can run on a laptop, but getting there was not always easy. People could make it work, but it often took more troubleshooting, more patience, and more time than it should have. For students, new users, and developers trying to use FreeBSD as part of their day-to-day work, that friction made a difference.

This was not just occasional feedback. We saw it come up in mailing lists, forums, blog posts, social media, and direct conversations with users. We heard it from commercial users too, especially those who wanted their developers to be able to build products on FreeBSD while also using FreeBSD on their own systems. We saw the same thing at the Foundation, with co-op students and interns trying to install FreeBSD on laptops and running into similar problems.

Over time, it became clear this was more than a recurring complaint. It was one of the things making FreeBSD harder for new users to adopt and harder for the community to grow.

> Yes, FreeBSD can run on a laptop, but getting there was not always easy.

Recognizing the Need

Once we stepped back and looked at the issue more broadly, it was clear that laptop support was an area where the Foundation could make a real difference. Better laptop support would not just improve the experience for current users. It could help students, developers, and new users get started more easily and make FreeBSD a more practical option for everyday use.

That matters because the systems people use every day often shape whether they stick with a platform, recommend it to others, or eventually contribute back to it.

Like many Foundation efforts, this did not start as one large, fully funded project. We started where we could and focused first on the problems that were causing the most frustration. Wireless was one of the clearest examples. Ask almost anyone what they most want fixed on a laptop, and wireless is near the top of the list. That made it one of the first areas we funded.

Turning a Broad Goal into Something More Concrete

Around the same time, the Foundation board was talking through long-term strategic goals and the kinds of work that could make the biggest difference for FreeBSD. One of those goals was increasing adoption.

That is a broad goal, and it only becomes useful when you turn it into something more concrete. One way we framed it was by asking a simple question: how do we make FreeBSD easier for a college student to use as a daily driver?

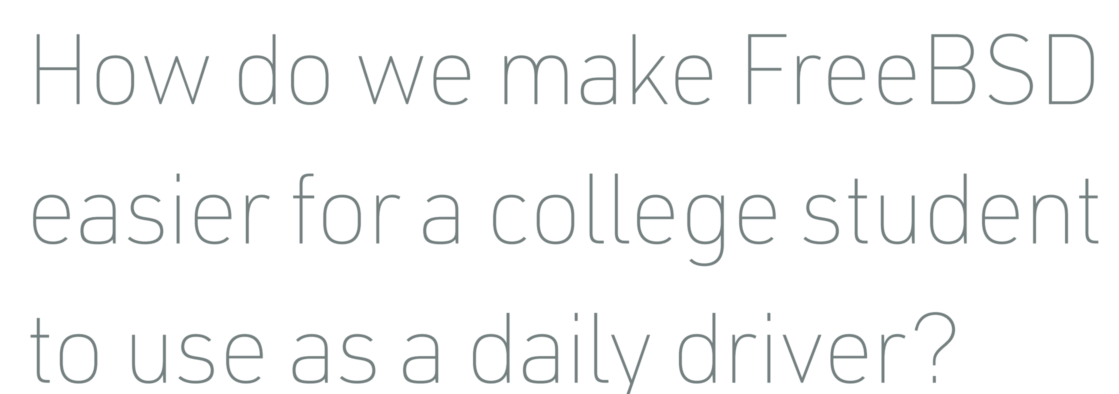
That gave us a much more practical way to think about the work. It also helped shape how we thought about laptop support. This was not just about fixing a few isolated technical issues. It was about improving the overall experience enough that FreeBSD would feel more usable on modern laptop hardware.

Building the Project

As we started scoping the work more fully, it became clear that the need was much greater than we could fund all at once. Early estimates put the cost at more than one million dollars. That was beyond what the Foundation could take on at the time.

Even so, we knew the work mattered, so we kept moving it forward where we could. We increased funding, brought on a userland developer, and added resources as they became available.

Then we got an email that changed the picture. It was short and straightforward. Someone wrote to say they loved FreeBSD, had been using it for more than 25 years, and wanted to help fund work to address some of the limitations they had encountered. That support became an important factor in allowing the project to move forward in a more substantial way.



The Laptop Support & Usability Project officially started in the fourth quarter of 2024.

What the Project Covers

From the beginning, the project focused on the areas that most directly affect the laptop experience: Wi-Fi, graphics, audio, the installer, and sleep states. Those are the things that make the difference between a system that technically runs and one that works well enough to use every day.

With 2025 as the project's first full year, the Foundation committed more than $750,000 to the work. That investment led to real progress.

In Wi-Fi, 2025 brought support for Wi-Fi 4 and 5 on key hardware, along with the start of Wi-Fi 6 work. Wi-Fi 4 and 5 drivers for Intel and Realtek are available in FreeBSD 15.0, with additional Realtek and Mediatek support still in progress.

Graphics support also moved ahead in important ways. Graphics drivers were upgraded to Linux 6.10, which is available in 15.0. Linux 6.11 is awaiting final review, and work on 6.12 is underway.

Audio saw important improvements in 15.0 as well. Users now have access to the new `sndctl(8)` and `mididump(1)` utilities, along with bug fixes, broader laptop support, and an initial effort to improve automatic sound redirection for HDA sound cards.

The installer improved in ways that matter to laptop users, too. In FreeBSD 15.0, the installer now supports downloading and installing firmware packages after the base system installation is complete. In 15.1, users can also install the KDE graphical desktop environment during installation.

Sleep states remain another major part of the project. Work on modern standby, or S0i3, is part of 15.1, while hibernate, or S4, remains part of the broader effort. That work also includes related areas, such as transitioning from modern standby to hibernate, handling disk encryption during hibernate, and improving sleep-state behavior for virtual machines.

The work continues in 2026 at a similar level of investment and scope. Current areas of focus include sleep states, graphics drivers up to Linux 6.18, Wi-Fi 6 support, USB4 and Thunderbolt support, HDMI improvements, UVC webcam support, and Bluetooth improvements.

A broader testing effort is also part of the work in 2026. Making individual components work is one thing. Testing how those pieces work together across a range of real hardware is another. That kind of testing matters if the project is going to be genuinely useful in practice.

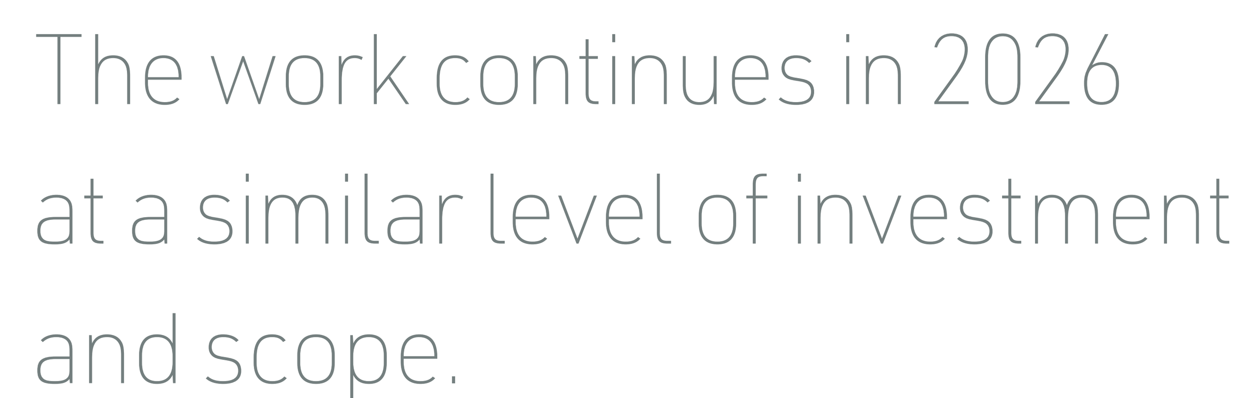
What Happens Behind the Scenes

One thing that can be easy to miss with a project like this is how much has to happen behind the scenes before the development work can really move forward. There needs to be a plan. Someone has to help manage the effort and keep it moving. We have to find developers with the right skills who are available and willing to do the work. As a nonprofit, we are always balancing what is needed against limited funding.

That is true for this project and for Foundation-funded work more broadly.

When we decide whether to fund a project, we look at a few basic things. Will it be impactful and beneficial to FreeBSD? Will it be useful and accessible to users? Does it need the kind of infrastructure and support the Foundation can provide? Do we have the funding to take it on responsibly?

That same reality applies to community proposals. Some move forward, and some do not. That is not because we do not care or are not interested. We know people put a great deal of time and thought into those proposals. With limited funding, we must ensure the work fits the bigger picture and has a meaningful impact.



The work continues in 2026 at a similar level of investment and scope.

Why This Work Matters

The Foundation's budget reflects these priorities. More than 62 percent of the current budget is allocated to software development that goes directly to improving FreeBSD. Another 17 percent supports advocacy and education, including outreach, storytelling, and community education. The Foundation also continues to invest in infrastructure, including the systems and hardware needed to support the Project over time.

The Laptop Support & Usability Project is not just about laptops. It is about making FreeBSD more usable, more approachable, and more practical for the kinds of day-to-day use cases that help bring people into the community and keep them there.

It is also a good example of how this kind of work usually comes together. The need is clear, the community keeps raising it, and over time, the funding, planning, and structure come together to make real progress possible.

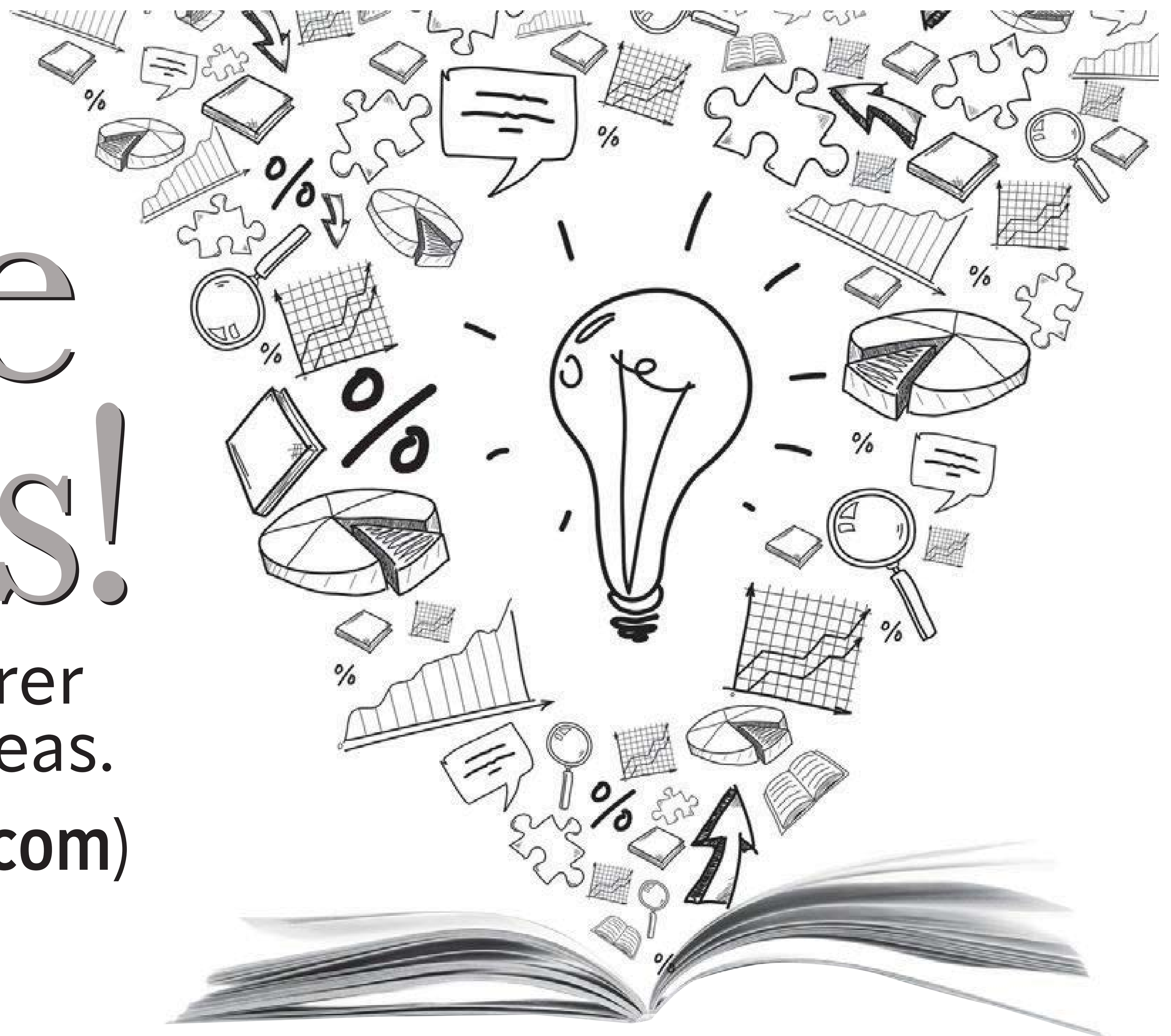
At its core, this project reflects what is possible when the community and the Foundation work together. The community kept raising the issues, sharing real-world experiences, testing what worked and what did not, and helping clarify where the need was greatest. The Foundation was able to turn that input into a funded, organized effort and help move the work forward. That kind of progress does not happen in isolation. It happens when community members, developers, donors, testers, and the Foundation all play a role in building something better together.

DEB GOODKIN is the Executive Director of the FreeBSD Foundation, joining as the first employee in 2005. When she's not running the Foundation or playing around with FreeBSD, you'll find her playing with her dogs, running on the backroads of Boulder, or reading a good book.

Write For Us!

Contact Jim Maurer with your article ideas.

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



Duplicating your System

Using duplicity to back up your FreeBSD desktop

BY JASON TUBNOR

You've just installed your new FreeBSD desktop (or laptop), got it just the way you like it, and are about to start work on that new novel or porting a new piece of software to FreeBSD that has just been released by your favourite software vendor. Then you realise that you're only one defective NVMe controller away from total data loss, taking all your work and time with it.

Backing up in modern times, we've had ZFS snapshots and replication to make this task extremely easy. However, you may not have access to another ZFS endpoint for replication, need to diversify risk by using a non-ZFS tool for backup, or are simply using UFS2, living the old skool life.

For these situations, my first recommendation is to lean on Tarsnap for its ease of use and simplicity, making restoration just as easy as backing up. But some situations call for a different approach. Maybe you have a strict firewall at your company that doesn't allow Tarsnap data streams to egress from your corporate network, or you have internal/easy access to storage endpoints, such as S3-compatible object storage or a large-file storage location with SFTP access.

Then you realise that you're only one defective NVMe controller away from total data loss.

When you are faced with the latter, the duplicity (sysutils/duplicity in ports) utility is available as an easily installable package onto your FreeBSD system:

```
# pkg install duplicity par2cmdline
```

In the above case, the par2cmdline package is also installed, as it isn't a dependency of duplicity but is used in this article (and by duplicity) to avoid bit rot in backup archives.

Duplicity (https://duplicity.gitlab.io/) will back up your various files and folders to encrypted tar-format volumes, while using librsync to improve compression and perform differential actions on data. While it doesn't have some of the really good features that make Tarsnap appealing, it does provide an alternative that works well for a lot of use cases.

While duplicity has many options for target storage, we are going to focus on using S3-compatible object storage here, as it is readily available from many cloud providers with varying cost vs access speed vs durability. The way duplicity works is more of an interactive method (especially when preening old data sets). While you might have success committing backups to AWS Glacier or similar object storage, restoration and management of data will be extremely slow, complex, and expensive; you have been warned. Don't aim for cold, cheap storage; go a little higher/warmer for a better experience. In saying it is interactive,

it will work perfectly fine when called from cron, executed within a script, or simply used on the interactive command line.

Before starting, we need to define the parameters and best practices for backing up data. Key considerations are:

- New full backup monthly — A long chain of incremental backups, while fast to back up, can blow out your Recovery Time Objective (RTO). Introducing frequent full backups will ensure the RTO window remains low.
- Encrypt your backup — You are backing up to someone else's computer. Unless you control the remote computer, data should be encrypted on your computer before it hits the network.
- Don't trust endpoint durability — Use the par2 function to incorporate parity blocks into your backup in the event of bit rot. The default is 10%, which can increase the cost for large archives. Assess your risk and adjust accordingly. We will use 5% in this example.

Before starting, we need to define the parameters and best practices for backing up data.

Prepare the AWS environment variables. These can go into a .duplicity.env file in your home directory. If you are on a multi-user system, ensure the .duplicity.env file is chmod 0600 to prevent other system users from gaining unauthorised access to private API keys:

```
~/.duplicity.env

export AWS_ACCESS_KEY_ID="<key_id>"
export AWS_SECRET_ACCESS_KEY="<access_key>"
export AWS_ENDPOINT_URL="https://sg-s3.storage.bunnycdn.com"

Then load our environment file for use:

$ . .duplicity.env
```

The final requirement before we begin using duplicity is to define a PASSPHRASE key for symmetrical encryption of your data when GPG is used (encryption is used by default). GPG Public key encryption can be used but is beyond the scope of this article (see the duplicity man page for further information). The PASSPHRASE variable can also be defined in the environment file we used above and then called from your backup script. For ease of documentation, it will be just exported here:

```
$ export PASSPHRASE="4640f58235c4ff7ad359fdcb5f2d8ac48f71c26bcff4a40f6d85503d0288a45e"
```

We are now ready to perform the first backup of our home directory from our freshly configured system:

```
$ duplicity backup --full-if-older-than 1M --exclude ./.cache \
      --par2-redundancy 5 ./ par2+s3:///mybucket-s3backup/computer
```

The above command engages the backup component of duplicity, creates a full backup if the previous full backup is older than one month, excludes the .cache directory, sets the par2 redundancy to 5%, taking all the files and folders from the current directory (the root

of the users home directory) and writes the backup to the mybucket-s3backup bucket in the /computer folder, using the par2 wrapper for block redundancy.

Once duplicity has completed its run, you'll be presented with the statistics for the backup:

```
Last full backup date: none
Last full backup is too old, forcing full backup
--------------[ Backup Statistics ]--------------
StartTime 1773460673.41 (Sat Mar 14 14:57:53 2026)
EndTime 1773460673.42 (Sat Mar 14 14:57:53 2026)
ElapsedTime 0.01 (0.01 seconds)
SourceFiles 15
SourceFileSize 6302 (6.15 KB)
NewFiles 15
NewFileSize 6302 (6.15 KB)
DeletedFiles 0
ChangedFiles 0
ChangedFileSize 0 (0 bytes)
ChangedDeltaSize 0 (0 bytes)
DeltaEntries 15
RawDeltaSize 6277 (6.13 KB)
TotalDestinationSizeChange 3884 (3.79 KB)
Errors 0
-------------------------------------------------
```

As this is a clean system, our home directory hasn't seen much activity yet and is quite small, so the above backup didn't take long to execute. The initial total that we have backed up:

```
$ du -sh .
62K    .
```

As you can see, even on a small dataset, 62KB of files (allocated disk space), 6.15KB is the actual file total with a destination size of 3.79KB after compression and encryption is taken into account.

Testing has shown that duplicity works as quickly as your hardware allows; the throttle will be set by how fast your object storage is and by the size of your network egress throughput. To observe how the backup is performing, use the `--progress` switch when executing duplicity on the command line.

At this point, your data is encrypted on remote storage and is unavailable, even to you, if you lose your symmetrical encryption key (PASSPHRASE). Take the time to put a copy of the key in a safe place in case you need to restore data to a fresh system.

Let's view what is in our first backup:

```
$ duplicity ls par2+s3:///mybucket-s3backup/computer/

Last full backup date: Sat Mar 14 14:57:52 2026
Sat Mar 14 14:00:35 2026 .
Sat Mar 14 13:33:22 2026 .boto
Sat Mar  7 15:55:14 2026 .cshrc
Thu Mar 12 22:15:04 2026 .gnupg
```

```
Thu Mar 12 22:15:04 2026 .gnupg/common.conf
Thu Mar 12 22:15:04 2026 .gnupg/private-keys-v1.d
Sat Mar 14 14:56:10 2026 .gnupg/random_seed
Sat Mar 14 14:00:35 2026 .lesshst
Sat Mar  7 15:55:14 2026 .login
Sat Mar  7 15:55:14 2026 .login_conf
Sat Mar  7 15:55:14 2026 .mail_aliases
Sat Mar  7 15:55:14 2026 .mailrc
Thu Mar 12 22:13:12 2026 .profile
Sat Mar 14 13:47:32 2026 .sh_history
Sat Mar  7 15:55:14 2026 .shrc
```

The list (ls) command in duplicity displays archives similar to **`tar -t`**. Again, here, the S3 connection has been made to mybucket-s3backup with the par2 wrapper to validate the consistency of each block and repair on the fly.

Time to start being productive in our home directory. Let's clone the ports branch to start with:

```
$ git clone --depth 1 https://git.freebsd.org/ports.git
Cloning into 'ports'…

$ du -sh .
1.0G    .
```

That has grown our home directory considerably. Now it is time to back up our work across multiple ports:

```
$ duplicity backup --full-if-older-than 1M --exclude ./.cache \
      --par2-redundancy 5 ./ par2+s3:///mybucket-s3backup/computer

Local and Remote metadata are synchronized, no sync needed.
Last full backup date: Sat Mar 14 14:57:52 2026
--------------[ Backup Statistics ]--------------
ElapsedTime 102.56 (1 minute 42.56 seconds)
SourceFiles 214568
SourceFileSize 682126695 (651 MB)
NewFiles 214554
NewFileSize 682120407 (651 MB)
RawDeltaSize 681815400 (650 MB)
TotalDestinationSizeChange 224163465 (214 MB)
-------------------------------------------------
```

As you can see, duplicity is extremely efficient. Our backup has only grown by 214MB, even though when the ports tree was checked out, our system reported 1GB.

Restoring a backup is just as simple as backing up:

```
$ duplicity restore --path-to-restore ports \
         par2+s3:///mybucket-s3backup/computer /tmp/temp/ports
Local and Remote metadata are synchronized, no sync needed.
Last full backup date: Sat Mar 14 14:57:52 2026
```

This command restores only the ports directory from the home directory that was previously backed up to the S3 bucket using the par2 wrapper. The destination of the ports directory will be /tmp/temp/

```
jtubnor@disk:~ $ du -sh /tmp/temp && ls -lsa /tmp/temp

1.0G    /tmp/temp
total 18
1 drwxr-xr-x   3 jtubnor wheel    3 Mar 14 16:56 .
9 drwxrwxrwt  16 root    wheel   16 Mar 14 16:57 ..
9 drwxr-xr-x  70 jtubnor jtubnor 82 Mar 14 15:18 ports
```

The output above shows that our 1GB ports directory has now been restored to the /tmp/temp directory. If you have had a complete machine failure, restoring your home directory is as simple as:

```
$ cd / && duplicity restore par2+s3:///mybucket-s3backup/computer /home/jtubnor/
```

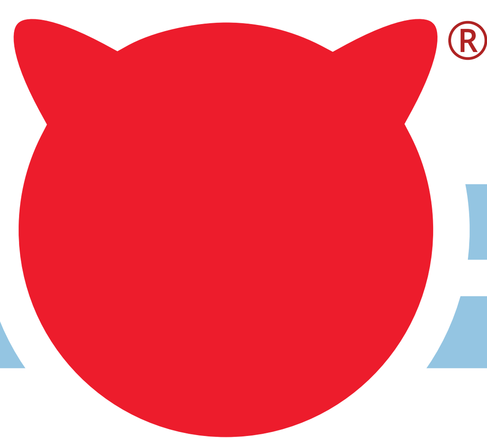
You need to ensure that you are currently not in the directory being restored to; this is why we have traversed to the root directory before issuing the restore.

This article barely scratches the surface of the many features contained within duplicity. It is even suitable for whole-system configuration backups, if needed, by adjusting as necessary. For more information on features and use case examples, check out the duplicity man page (https://duplicity.gitlab.io/stable/duplicity.1.html)

Duplicity is a small, mature program with only a few dependencies, but it works well in most situations and is easily installable on any FreeBSD system.

JASON TUBNOR has over 30 years of IT industry experience in a vast range of disciplines and is currently the Manager of Technology Operations at Latrobe Community Health Service (Victoria, Australia). Discovering Linux and Open Source in the mid 1990s, then being introduced to OpenBSD in 2000, Jason has used various BSDs to solve problems in organisations that cover different industries. Jason is also a co-host on the weekly BSDNow Podcast (https://bsdnow.tv).

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

Adventures in TCP/IP

Improving TCP's Responses to Reordering

BY RANDALL STEWART

This column will be slightly different from the others in this series. It describes a set of problems that arose with the use of TCP and provides a walk-through of how the problems were troubleshooted, illustrating information covered in previous columns. It will also refer to some TCP mechanisms that were covered in previous columns in this series. It will conclude by tying it all together and showing not only how the problem was solved, but also how the RACK stack became more resilient to packet reordering due to a driver bug that was found and fixed.

He expected Mbps speeds but instead got extremely slow downloads.

The Problem

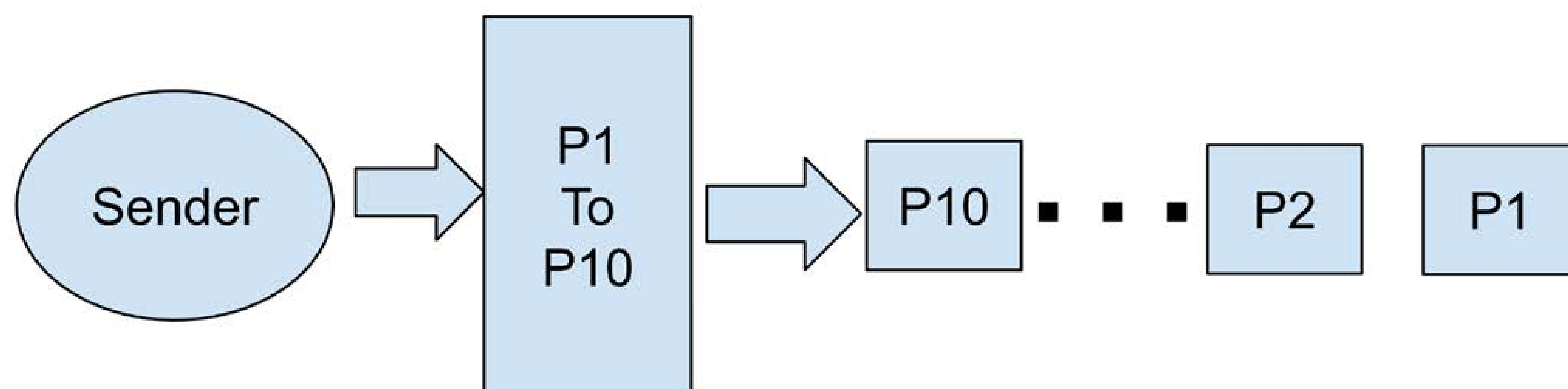
So, sometime last year, Drew Gallatin approached me with a TCP problem he was having. It started with him attempting to download a set of packages to be updated in his new location with the FreeBSD package tool (`pkg`). And things were miserably slow. He expected Mbps speeds but instead got extremely slow downloads. So, he started testing with another FreeBSD site he has access to. This led him to try changing to the RACK stack instead, since it had performed better for him in the past. And sure enough, as he told me, he was getting three times the performance with the RACK stack as with the FreeBSD default stack. But three times 10 kbps is only 30 kbps, and he was expecting at least two orders of magnitude better performance than that. He passed this problem to me, asking for help as to why TCP in both the RACK stack and the FreeBSD stack was performing so badly on his 'high-speed' Internet.

Identifying the Problem

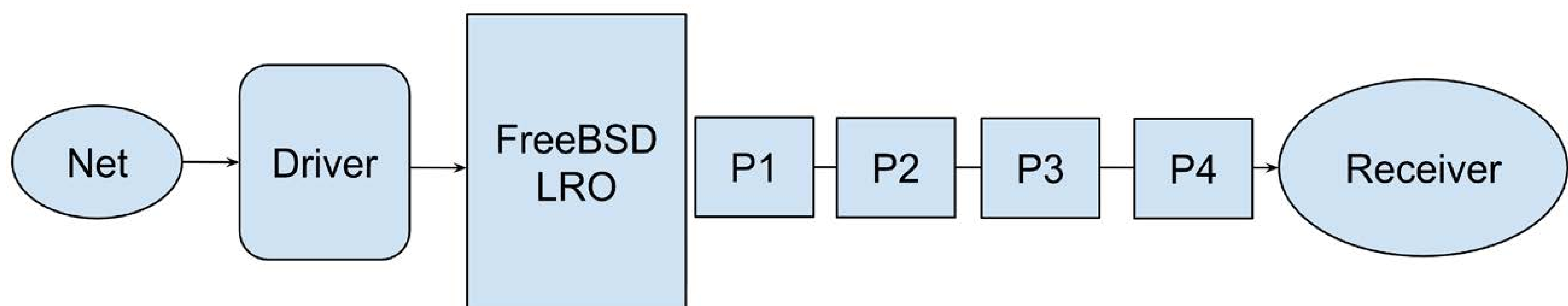
As discussed in the column on Black Box Logging [1], Black Box Logging (BBlog) provides detailed debugging, especially for the RACK stack. The very first thing to do was to enable BBlog on both the sender and receiver. Please also refer to that column for detailed descriptions of how to enable BBlog.

For Drew's situation, I ended up providing him with a couple of custom client-server programs that would send a file and discard it but also enable BBlogs. Drew then ran the BBlog

collector (`tcplog_dumper`) and the specific programs to send a comparably sized file, similar to what he was trying to download to his new location. Once he had the BBlogs, he allowed me to read and interpret them. I won't bore you with the copious text that the BBlog entails, but they quickly painted a picture of the sender and receiver that immediately told what was happening at the TCP layer. Basically, the sender would begin with its initial window of roughly 10 MSS (14,600 bytes) and send a burst to the internal driver TSO mechanism:



The data packets would cross the Internet and arrive at the TCP receiver as follows:



The receiver was sending SACKs and ACKs, indicating that the packets were processed by the receiver in the order P4, P4, P2, and finally P1.

Now, from the BBlogs, I could not determine where the reordering was coming from — you are only seeing TCP's perspective after all. What is happening below the FreeBSD LRO layer (i.e., the driver and network) is not visible in the BBlogs (note that some LRO logs are available in BBlogs). It is also not uncommon to see reordering in a network, as shown by Bennett, Partridge, and Shectman [2]. So, I asked Drew to investigate why his network was experiencing massive reordering — and it was quite massive.

Now, from the BBlogs, I could not determine where the reordering was coming from – you are only seeing TCP's perspective after all.

It appeared that every group of 4-5 packets would arrive in the reverse order in which they were sent. So, you would see your burst of 10 initial packets translated into an arrival pattern of P4, P3, P2, P1, followed by P8, P7, P6, P5, followed by P10 and P9. This behavior will cause the FreeBSD TCP stack to go into recovery constantly since it sees three duplicate ACKs/SACKs, which start retransmitting just as the ACK for all the packets arrives. Once you enter fast recovery, as discussed in [3], you exit the recovery after all in-flight data is acknowledged and end up with a congestion window cut in half and the `ssthresh` point (the point at which we flip from slow start to congestion avoidance) set to that same value.

This meant the FreeBSD TCP stack could never really get out of recovery for very long, which easily explained the horrible performance Drew was seeing. Once recovery ended, it would drop back into congestion avoidance. Anytime the congestion window allowed a

four-packet burst, the stack would once again see three duplicate ACKs/SACKs and reenter recovery.

The Root Cause of the Issue

Drew, a FreeBSD developer (who works with drivers for fun), found after a little investigation that it was actually an error in the driver interface to LRO that caused the reordering. Basically, the hardware had multi-queue disabled, so that the hardware stopped calculating the RSS hash because it was not needed. The driver, however, was still marking the RSS hash as valid. The FreeBSD LRO code uses the RSS hash to sort the received packets. This mis-marking thus caused LRO to sort the packets incorrectly. Once his driver was patched to send packets in the correct order to LRO, both RACK and the FreeBSD stack's performance improved massively, on the order he was expecting. It was great helping Drew find this issue using BBlogs, but the whole incident raised more questions for me from a TCP perspective. And it may even spark a question in your mind, too, if you are unfamiliar with the RACK stack.

Further Questions Generated by the Bug

Why did the RACK Stack Perform Better than the FreeBSD Stack?

For those of you unfamiliar with RACK, you may wonder why RACK performed three times better than the FreeBSD stack (even though that was only 30 kbps, it was still quite a difference). This is because, as discussed in [4], the RACK stack has protections against reordering. It does this by using a timer in combination with loss reports. When a SACK arrives telling that data is lost, it will wait to retransmit if not enough time has elapsed. The timer is usually the latest RTT plus a small extra delay, but as RACK sees reordering in the network through DSACKs (a DSACK is a SACK indicating that a TCP segment was received more than once by the receiver), it will expand this timer up to two times the last RTT seen. This meant that after the initial recovery, where the extra delay was small, the arrival of DSACKs increased the RACK delay timer enough that it would not constantly go into recovery but would instead wait long enough for all acknowledgments to flow back.

You may wonder why RACK performed three times better than the FreeBSD stack.

Why was the RACK Stack Still So Slow?

For me, this was the fundamental question posed by this bug. RACK has this wonderful mechanism to protect from reordering, so why, after its initial recovery, did RACK not quickly speed up to reach Drew's anticipated megabits per second? With a bit of digging into the BBlog, I ultimately discovered the answer.

This question goes back to the fundamentals of the way congestion control works. Once you exit the initial slow start with a loss/recovery incident, your `ssthresh` is set to 5 packets (the initial 10 cut in half). This then means that RACK is forever in congestion avoidance. And this places a significant drag on TCP. When in congestion avoidance, you increase your congestion window by one MSS every time a full congestion window of data is acknowledged. So, you send and have acknowledged five packets and now raise your congestion window to six packets. Now you send and get acknowledged six packets, and then raise it to seven. This

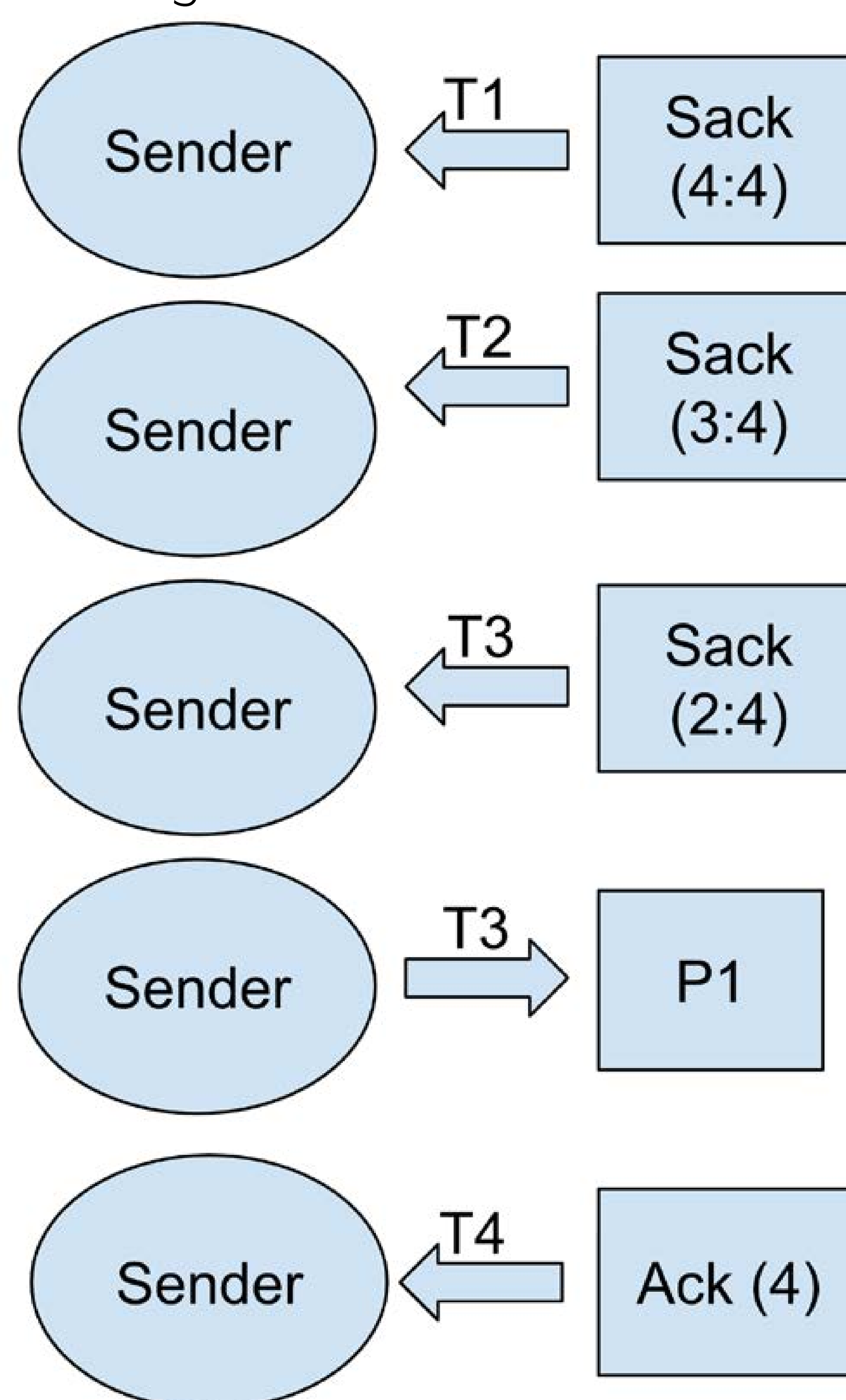
continues until some other error occurs or the connection data is all sent, and the connection completes.

So, in looking at his round-trip time (about 45 ms or so), that means that every 45 ms you would grow the congestion window by one packet. That sounds fine until you realize that to fully utilize a 1,000 Mbps network, your congestion window at 45 ms RTT needs to be open to about 3,850–3,900 packets. So, you would need around 380 round trips for RACK to start using his network fully, which is about 170–180 seconds. If he had a single super-large transfer, this would not be too noticeable (in the grand scheme of things), but each time a transfer was completed, it would start a new TCP connection, once again hitting the same issues. This made his 'fast' network horribly slow in his perception.

So, this raised a fundamental question in my mind. How can we change the RACK stack to better recover from this situation?

What Improvements Are Now Integrated Into RACK Stack?

So, what I realized was a way for RACK to 're-enter' slow start (note that slow start is a misnomer in my opinion, since it's exponential growth, doubling the congestion window every RTT) so that it could open its window much faster. So, when looking at the initial situation, RACK would see the following:



The key here is that time mark T3 (in comparison to the transmit time) is just enough time longer than the RTT to tell RACK that yes, it's time to retransmit the packet. This is because RACK had not, to this point, seen any duplicate acknowledgments, and the initial send to T3 was longer than an RTT plus the small initial delta RACK uses. But even more importantly, T4 — the actual arrival of the acknowledgment that moves the cumulative point to P4 — happens in much less time than an RTT. Basically, the time between T4 and T3 was always less than half of the RTT. This provided an insight into how RACK could be adjusted to handle this scenario.

RACK now tracks how much data, in bytes, has been retransmitted. Any time a SACK or ACK arrives that has had only one retransmission, and the acknowledgment (SACK or ACK) is less than half of the SRTT from when the retransmission was sent, it will reduce that count. If the count reduction is from an ACK (where the cumulative acknowledgment point is moving forward) and the count has fallen to zero, this then means that we have hit a situation where the actual entry of recovery is false, and we should restore the previous `ssthresh` and congestion window and resume slow start.

Adding the above small change to RACK changes its poor performance to match closely what one would see with no reordering in the picture and provides the RACK stack with even better protection from reordering.

1. R. Stewart, M. Tüxen, "Adventures in TCP/IP: TCP Black Box Logging", in *The FreeBSD Journal May/June, 2024*.
2. C. R. Bennett, C. Partridge and N. Shectman, "Packet reordering is not pathological network behavior," in *IEEE/ACM Transactions on Networking*, vol. 7, no. 6, pp. 789-798, Dec. 1999.
3. R. Stewart, "Dynamic Goodput Pacing: A New Approach to Packet Pacing", in *The FreeBSD Journal November/December, 2024*.
4. R. Stewart, M. Tüxen, "RACK and Alternate TCP Stacks for FreeBSD", in *The FreeBSD Journal January/February, 2024*.

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

The FreeBSD Project is looking for

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

Find out more by

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

Downloading the Software
freebsd.org/where.html

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

Already involved?

Don't forget to check out the latest grant opportunities at **freebsdfoundation.org**

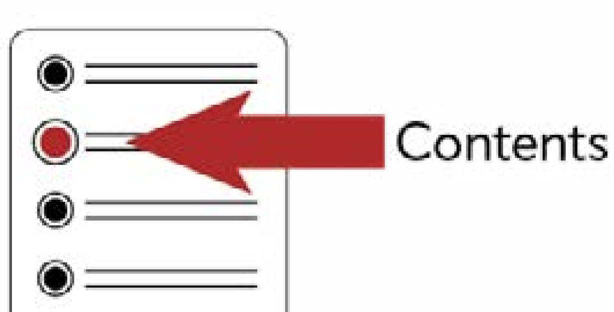
Help Create the Future. Join the FreeBSD Project!

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

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

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

The FreeBSD Community is proudly supported by



Events Calendar

BSD Events taking place through June 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.

Hackathon 202604

April 24 – 26, 2026
Wiesbaden, Germany
https://wiki.freebsd.org/Hackathon/202604

We're pleased to share that planning is underway for a regional FreeBSD Hackathon in Wiesbaden this April. The event is designed to bring together local community members, but participation is open to anyone interested in contributing to FreeBSD. The goal is to collaborate, connect, and make meaningful progress on FreeBSD in a focused and supportive environment.

Open Source Summit North America 2026

May 18 - 20, 2026
Minneapolis, Minnesota
https://events.linuxfoundation.org/open-source-summit-north-america/

The summit features a wide range of tracks covering Linux, cloud and containers, AI, security, embedded systems, and more. Attendees can expect keynote sessions, technical talks, and opportunities to connect with the broader open-source community. Deb Goodkin, Executive Director of the FreeBSD Foundation, will be speaking at the conference.

Ottawa 2026 FreeBSD Developer Summit

June 17-18, 2026
Ottawa, Ontario, Canada
https://wiki.freebsd.org/DevSummit/202606

Co-located with BSDCan 2026, the two-day event consists of developer discussion sessions, vendor talks, and working groups.

BSDCan 2026

June 17 - 20, 2026
Ottawa, Canada
https://www.bsdcan.org/2026/

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