FreeBSD® JOURNAL
September/October 2021

FreeBSD Code Review with git-arc

How to Implement a Simple USB Driver for FreeBSD

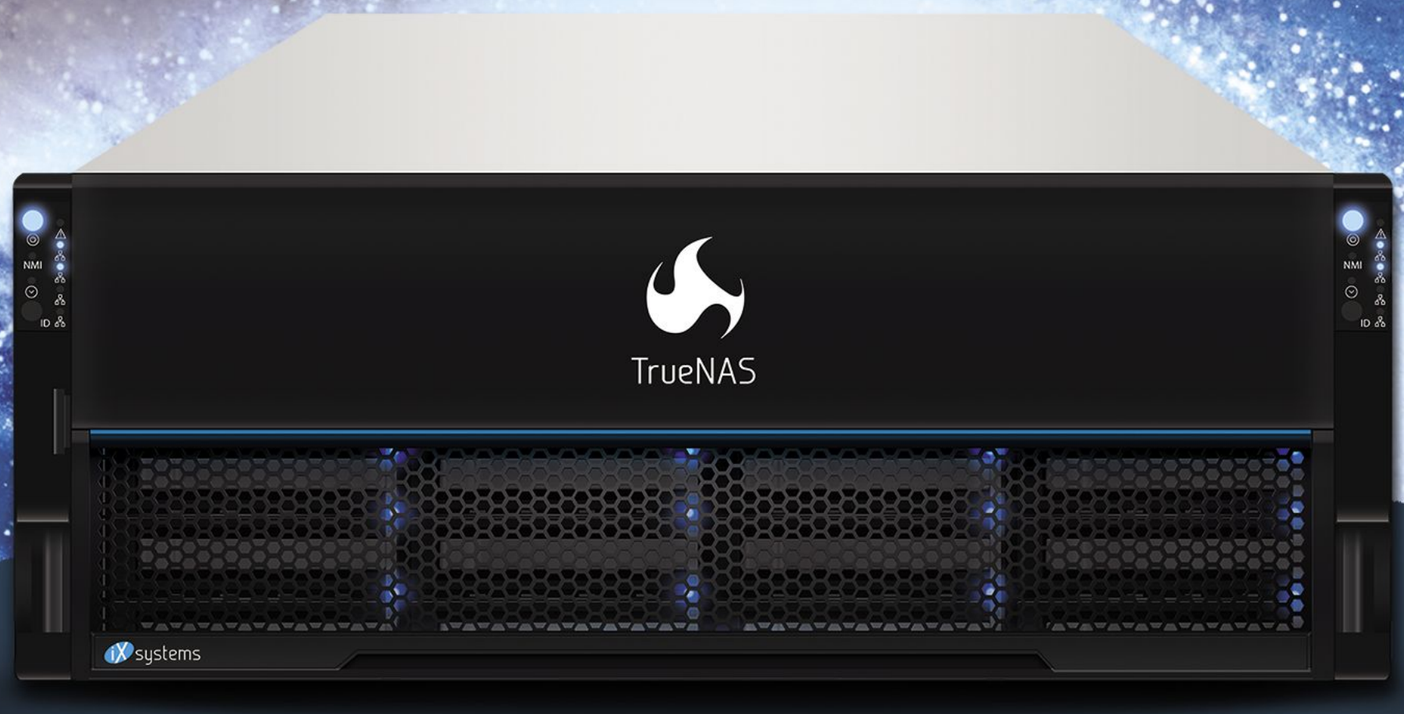
Kernel Development Recipes

Practical Ports

Programmers Programming Potpourri

TrueNAS® M-SERIES
Powerfully Scalable Enterprise Storage

OPEN STORAGE FOR ENTERPRISE WORKLOADS



UTILIZES FLASH-OPTIMIZED ZFS TECHNOLOGY

IDEAL FOR LATENCY-SENSITIVE AND BUSINESS-CRITICAL VIRTUAL MACHINES AND PHYSICAL WORKLOADS.

- PERFORMANCE AND SCALE WITHOUT COMPROMISE
- INTELLIGENT STORAGE OPTIMIZATION
- SELF-HEALING DATA PROTECTION
- UNLIMITED SNAPSHOTS AND REPLICATION

Contact iXsystems to Learn More about what TrueNAS® can do for your business!

iXsystems.com/TrueNAS | (855) GREP-4-iX

iXsystems™

FreeBSD Journal (ISBN: 978-0-615-88479-0) is published 6 times a year (January/February, March/April, May/June, July/August, September/October, November/December).
Published by the FreeBSD Foundation,
3980 Broadway St. STE #103-107, Boulder, CO 80304
ph: 720/207-5142 • fax: 720/222-2350
email: info@freebsdfoundation.org

LETTER from the Foundation

Welcome to the September/October issue of the FreeBSD Journal. The fall season means a lot of things to the Foundation team. Beautiful weather in Boulder, CO where the Foundation is headquartered, end of year planning, and of course, kicking off our Fall Fundraising campaign. So far, the responses I'm hearing are extremely positive, and I'm hopeful that we will reach our fundraising goal this year.

As of this writing, we've raised $200,000 towards our $1,250,000 goal for 2021. Why do we need so much money? Well, last year we decided to make more significant software contributions to FreeBSD. In order to do that, we had to grow our team. We developed a technology roadmap from input we were receiving from individual and commercial users as well as market trends. Based on the roadmap, we identified positions we needed to fill.

This year we've hired three full-time software developers, one full-time Arm kernel developer, and one project manager. We also are funding wifi work full-time and some other projects to help with FreeBSD on the desktop. We are doing this to help attract new users and contributors to the Project. More on those projects can be found in the upcoming FreeBSD Project Q3 Status Report.

Our growth wasn't just in our technology team, but in our advocacy team too. We hired a marketing coordinator and technical writer to provide more educational and informational content.

We also continue to fund this very journal, making it free to everyone. We feel the *FreeBSD Journal* is an important piece of our advocacy puzzle. The current issue focuses on FreeBSD Development and includes articles on FreeBSD Code Review with git-arc, FreeBSD Development Recipes, and more. We hope you enjoy these articles and share them with your friends and colleagues.

Finally, as we continue our Fall Fundraising campaign, please remember that the Foundation is 100% funded by donations. I know we say it a lot, but we truly can't do it without you. Please consider making a donation to help us continue and increase our support for FreeBSD.

Happy Reading!
Deb Goodkin
FreeBSD Foundation Executive Director

FreeBSD® JOURNAL
September/October 2021

FreeBSD Development

WeGet letters

by Michael W Lucas

**Greetings and felicitations,
oh mighty Letters Column Master!**

Now that I've gotten the obligatory "sucking up so you'll pay attention" out of the way, I'll ask: are you crazy? You're supposed to be answering people's sincere and heartfelt letters, and instead you tell them that they're doomed for even asking. This is an issue dedicated to development, and I bet anything you're going to spend your pages slagging on developers.

How dare you, sir? How *dare* you?

—Not A Fan

Dearest NAF,

I have absolutely nothing against developers. Most—er, many of them are lovely human beings. I simply wish that they had dedicated their lives to something that might improve civilization, like volunteering to pick up trash by the roadside.

My problem is with code, not coders.

We treat computer code like a precious treasure worthy of hoarding, when in reality it's like nuclear waste with a few rubies scattered in it. While every line that emerged from the CSRG is unalloyed platinum, most code repositories contain a whole bunch of barely functional spew supporting occasional scintillating scraps of brilliance. Some of those luminous lines are shackled into supporting the great threats dooming our civilization, like Facebook.

Yes, the world—really the Internet, but if you're a developer isn't the Internet your entire world?—is bloated with documentation on how to write better code, but none of it agrees with one other and most of you can't be bothered to read the instructions anyway. No, don't argue. I write that documentation; I have nearly unholy knowledge of how many of you read the stuff.

If you want to be a developer and yet improve civilization, use your hard-won acumen towards *reducing* the amount of code the world uses.

Every line of code is a seed of technical debt waiting for an opportunity to sprout into a malignant blossom, and every program is a farm of their horrific sprouts. Every package you install begins suffering from neglect the instant you log out, which is why some of you have terminal sessions that have been open for six years and think it's okay because the server is behind the firewall, and we're all doomed anyway. Computer people always think that there's a technical solution when the only solution is to shut off the laptop and hang out in meatspace for a lifetime or two.

Very few developers spend their careers writing clean, new, perfect implementations. The university churns out these bright-eyed maniacs who think that they'll be writing IP routers in Java just like their senior project demanded, then they get a job where they're tipped face-first into the nuclear waste vat and told to make it not radioactive. They spend aeons fixing bugs caused by other people's insufficient grasp of how their code works, until they achieve enough seniority that they're allowed to write their own bugs.

It's enough to make someone clean-field write a nearly useless program in the hope of demonstrating what good software should look like and post it on Github, just to prove that they exist. Or that they used to exist. Did you know Github has a feature to set the heir to your code? That horrible program you wrote for your own satisfaction, but other people discovered and filed bugs against until it took over your life and finally made you stroke out? Before you retire and start choosing which brand of dollar store cat food you'll be dining on for your twilight years, be sure to choose your code's next victim. If you pick me, I'll immediately auction off all rights to the least savory bidder and exploit the proceeds to soil your legacy.

The most heroic developers are those who delete code.

So much code hasn't been touched in decades because it seems to work, when the reality is it's failed in ways nobody has noticed yet. Study it. Should it be ripped out because it's old? Certainly not! It should be ripped out only if and when there's a more maintained method of doing the same thing.

Probably a library. One of my least loathed "innovations" of the last couple decades is FreeBSD's libarchive.

Unix has too many formats for compressing and collating data because most of them were invented on and for Unix. Does anyone with less than a decade of experience understand when to use compress(1) versus Microsoft CAB archives? No, because nobody with any amount of experience remembers that except for a few hard-core archive format specialists. What about the hydra-headed tar format? Eliminate one tar format, and two more grow to take its place. Worse, each of those new tar formats are optimized for increasingly particular use cases.

Every archiving program supported its own format. Many of them had marginal support for other formats. When I started as a sysadmin I could use tar(1) to unzip archives, except when the zip format was really compressed and some Idiot (me, I'm Idiot) slapped the wrong extension on the filename.

Libarchive provided a single central source of compression and archiving truth. Programs that relied on libarchive could work with any file format. Bugs discovered and fixed in libarchive instantly propagated to every program that linked it.

The real benefit of libarchive was that it reduced the amount of code in use.

Instead of dozens of programs sketchily implementing their own so-called support of whatever formats they preferred, these programs discarded their own engines and pulled in libarchive. This library might have tens of thousands of lines of code, but using it removed hundreds of thousands of lines of code. Plus, it let sysadmins use their preferred archiving tool to open anything. Early in my career, I learned to be comfortable using tar(1) in the some way certain circus performers are comfortable slipping a tractor/trailer tow chain up their nose and out their ear. Today, I use tar to open those pesky CAB files that are such a burden on sysadmins.

Meanwhile, GNU tar still relies on file extensions.

I don't know how Linux people cope. Maybe that's why they so fiercely cuddle their penguins.

Can libraries be taken too far? No. Only vision can fail. Why, one night at BSDCan a few FreeBSD developers who'd had more liquor than sleep had the spark of genius to implement and

publish libtrue, a back-end to the true(1) program that could be linked into any program. Sadly, the world failed to pick up on this magnificent innovation and libtrue remains underadopted.

If you want to be a developer and make the world better, study your nuclear waste with an eye towards reducing it. Does it have ancient functions that can now be served by a well-maintained—mostly maintained—er, maintained at all, in any way—library? Are there common features that should be in a library?

How can you reduce the amount of code in the world?

Because code is unquestionably poison. Just look at what it's done to you, making you question my ethics when it's obvious I don't care.

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

MICHAEL W LUCAS is the author of *TLS Mastery*, *Absolute FreeBSD*, and the *$ git commit murder series*. His *DNSSEC Mastery* and *Domesticate Your Badgers* should be out first thing in 2022, and it's far too late for you to stop him. Submit your questions to letters@freebsdjournal.org.

FreeBSD Code Review with git-arc

BY JOHN BALDWIN

The FreeBSD Project uses the Phabricator Differential application as a tool for code review at https://reviews.freebsd.org. Phabricator itself provides several applications to support software development, but the FreeBSD Project only uses the code review tool. Users and developers can upload changes for review either by pasting diffs directly into the web application, or by using the **arc** command line tool (available via the devel/arcanist package or port). arc can upload commits from a git branch from the command line and can also modify commit logs of reviewed commits to prepare them for pushing to the public repository.

However, **arc** has a few limitations that make it awkward to use for FreeBSD development:

- Arcanist uses its own commit log template whose format does not match FreeBSD's existing template and is not easily changed.
- Arcanist presumes a model where all the commits in a development branch are uploaded for review as a single Differential revision. When working on a feature branch with multiple commits, it is usually more efficient to review each commit individually.

While these limitations can be worked around (for example, individual commits can be uploaded as separate reviews by careful use of the `--head` option), the work arounds are tedious.

The git-arc tool provides a wrapper around **arc** that mostly mitigates those limitations. git-arc is a plugin for git which adds commands to work with Phabricator.

Installing git-arc

git-arc lives in the FreeBSD source repository at `tools/tools/git`. The script and its manpage can be installed by running make install from that directory. Note that git-arc is installed to `/usr/local/bin` by default.

```
> git clone https://git.freebsd.org/src.git
...
> cd src/tools/tools/git
> make
gzip -cn /usr/home/john/work/freebsd/main/tools/tools/git/git-arc.1 >
git-arc.1.gz
> make install
install  -o root  -g wheel -m 555
/usr/home/john/work/freebsd/main/tools/tools/git/git-arc.sh
/usr/local/bin/git-arc
install  -o root -g wheel -m 444 git-arc.1.gz  /usr/share/man/man1/
```

In addition, git-arc requires the following packages to be installed: arcanist (devel/arcanist), git (devel/git), and jq (textproc/jq).

Since git-arc is a wrapper around arcanist, arcanist must also be initialized. First, create an account at https://reviews.freebsd.org. Second, install an API token for arcanist:

```
> arc install-certificate https://reviews.freebsd.org
 CONNECT  Connecting to "https://reviews.freebsd.org/api/"...
LOGIN TO PHABRICATOR
Open this page in your browser and login to Phabricator if necessary:

https://reviews.freebsd.org/conduit/login/

Then paste the API Token on that page below.

    Paste API Token from that page: cli-XXXXX
Writing ~/.arcrc...
 SUCCESS!  API Token installed.
```

Using git-arc For a Single Commit

The first step in reviewing a commit is preparing the commit for review. The commit should be a commit candidate with a suitable log message. Any fixups should be squashed back into the commit so that the commit matches what would be pushed to the tree.

Creating the Review

Once the commit is prepared, the next step is to create a review for the commit via the create subcommand. This command accepts a reference to the commit (e.g., a hash, or a symbolic reference such as HEAD) as a positional argument after any options. The create subcommand creates a new review in Phabricator using the commit's log message to set the review title and description. Reviewers can be added via the `-r` option. Multiple reviewers can be specified as a comma-separated list or via multiple `-r` options. A group can be added as a reviewer by prefixing it with the '`#`' character, e.g. `#bhyve` to tag developers working on the bhyve(8) hypervisor. This example creates a review for a commit on the **gdb_11** branch to the `devel/gdb` port marking the port maintainer (pizzamig@FreeBSD.org) as a reviewer:

```
> git log --oneline main..gdb_11
6b21ea620990 devel/gdb: Update to 11.1.
> git arc create -r pizzamig gdb_11
commit 6b21ea62099007a0d376852731cdbde4d8d522d9 (HEAD -> gdb_11)
Author: John Baldwin <jhb@FreeBSD.org>
Date:   Thu Sep 16 17:25:13 2021 -0700

    devel/gdb: Update to 11.1.

…
Does this look OK? [y/N] y
...
```

The `create` subcommand first displays the commit including both the log message and the patch. It then displays a prompt confirming if a review should be created. Answer '`y`' to continue and create the review.

Updating the Review

The update subcommand can be used to update the review after changes are made to the patch. First apply any changes to the git commit in question either by amending the commit or by committing new changes and then squashing them back into the original commit. Note that the update command only updates the patch in the review. It does not update the review's title or description, nor does it permit adding additional reviewers or subscribers. Those changes must be made in Phabriactor's web application instead.

Note: The update and stage subcommands use the first line of commit logs to find a review with an identical title. If the first line of the commit log message is changed, the title of the review must be updated in Phabricator before git-arc will properly recognize the existing review for a commit.

This example updates the review for the devel/gdb port update after amending the original commit with fixes from reviewer feedback:

```
> git arc update gdb_11
commit 8d532ac873633ae12d42be709755f56f9d86c310 (HEAD -> gdb_11)
Author: John Baldwin <jhb@FreeBSD.org>
Date:   Thu Sep 16 17:25:13 2021 -0700

    devel/gdb: Update to 11.1.

...
Does this look OK? [y/N] y
...
```

As with the `create` subcommand, update displays the log message and patch followed by a prompt to confirm the update. Answer '`y`' to continue and update the review. Next, an editor window (using the editor configured in the user's `$EDITOR` environment variable) will open. Enter a description of the changes made for this update, save the file, and exit the editor. This description will be added to the review as a comment.

Finalizing the Review and Pushing the Commit

Once the commit is ready to be published, the `stage` subcommand merges the commit to the local main branch and amends the commit log with metadata from the review. Specifically, `git arc stage` adds properly formatted 'Reviewed by' and 'Differential Revision' tags to the commit log. After the operation completes, it leaves the working tree set to the new tip of the main branch including the staged commit. The commit can then be reviewed via `git show` before pushing to the public tree. This example shows the final push of the update to `devel/gdb`:

```
> git checkout main
> git fetch freebsd
```

```
> git merge freebsd/main
> git arc stage gdb_11
> git push freebsd
```

The first three commands ensure the local main branch is up to date before staging the commit. The stage subcommand pops up an editor window with the updated commit log after merging the commit. This provides an opportunity to fix any formatting issues in the commit log. Save the commit log and exit the editor to continue.

Note that the `stage` subcommand fails if it encounters conflicts when merging the commit onto `main`. To resolve, rebase the original commit onto the updated main branch. If the changes to fix the conflict resolution warrant review, then update the review. Otherwise, re-run the stage subcommand with the rebased commit.

Using git-arc For a Branch

While git-arc is useful for individual commits, it provides the greatest benefit when working with a branch containing multiple commits. The `create`, `update`, and `stage` sub-commands all accept multiple commits in a single invocation. Commits can be identified either via individual references as in the single commit examples above, or as Git revision ranges.

Creating the Reviews

For branches, git-arc creates reviews for each commit. These commits are linked together into a **stack** in Phabricator. The review for the first commit in the branch is marked as a parent revision of the review for the second commit and so on. This permits all of the branch commits to be found in the Phabricator UI from the Stack tab when viewing a review for any individual commit in the branch.

By default, the create subcommand will display the log message and patch for each commit, prompting after each commit. For a branch with many commits, this step can be tedious, so `create` accepts an optional `-l` argument. If this argument is given, then create will list all of the candidate commits with a single confirmation prompt. If the user answers '`y`' at the prompt, then the stack of reviews are created without further prompts. This example creates reviews for all of the commits for a branch checked out in the current directory. The branch was created as a branch off of `main`:

```
> git arc create -l -s emaste main..
2f7e09973ab6 cryptodev: Use ‘csp’ in the handlers for requests.
fbc805bb4d62 ccp, ccr: Simplify drivers to assume an AES-GCM IV length of 12.
8390644fd45f crypto: Permit variable-sized IVs for ciphers with a reinit hook.
f08d44eaa9ee cryptosoft, ccr: Use crp_iv directly for AES-CCM and AES-GCM.
43aaceb5afef cryptodev: Permit explicit IV/nonce and MAC/tag lengths.
0190b9e740d8 cryptodev: Permit CIOCCRYPT for AEAD ciphers.
03f07b455c80 cryptodev: Allow some CIOCCRYPT operations with an empty payload.
e0caaccf1ec0 cryptocheck: Support multiple IV sizes for AES-CCM.
ae4a0338bc8b crypto: Support multiple nonce lengths for AES-CCM.
cb18504c7712 aesni: Support multiple nonce lengths for AES-CCM.
60e1a45f2201 aesni: Handle requests with an empty payload.
aa653be04078 aesni: Permit AES-CCM requests with neither payload nor AAD.
fafcdb583930 aesni: Support AES-CCM requests with a truncated tag.
75c003b9ccbb ccr: Support multiple nonce lengths for AES-CCM.
```

```
de32cc23b0f9 ccr: Support AES-CCM requests with truncated tags.
6a88ac41d972 safexcel: Support multiple nonce lengths for AES-CCM.
ff4f260a5fd9 safexcel: Support truncated tags for AES-CCM.
e46422be0eaf cryptosoft: Fix support for variable tag lengths in AES-CCM.
d98f930cd833 crypto: Test all of the AES-CCM KAT vectors.
7ee5d373884e crypto: Support Chacha20-Poly1305 with a nonce size of 8 bytes.

Does this look OK? [y/N] y
...
```

If one wishes to create individual reviews for commits on a branch without linking the reviews together, that can be done by invoking `git arc create` separately for each commit. The relationship between reviews can also be adjusted in the Phabricator web interface.

Checking Review Status

When working with a branch, it can be useful to examine the status of the individual reviews associated with a branch. This can be done via the `list` subcommand. The list subcommand accepts one or more commit names or commit ranges and outputs a single line of status for each commit. For commits without an associated review, the status is reported as `No Review`.

This example shows the status of several commits on a `gcc9_universe` branch. For this particular branch, the author has chosen to review commits individually rather than as a series. As such, some commits are not yet ready for review while others have been approved for merging to `main`.

```
> git arc list main..gcc9_universe
f0f665a2f4a5 Accepted     D26202: Switch to GCC 9 for the GCC tinderbox.
a98a78e2dabc Accepted     D26203: Pass -msecure-plt to GCC for 32-bit powerpc.
b4412a18ab23 Needs Review D31933: hyperv storvsc: Don't abuse struct sglist to hold virtual addresses.
a0eaf413441a Accepted     D31934: kernel: Disable errors for -Walloca-larger-than for GCC.
daf618e9a8c4 No Review    : Fix various places which cast a pointer to a vm_paddr_t or vice versa.
8c46bb47a57f Needs Review D31938: bhyve: Add an empty case for event types in mevent_kq_fflags().
46d2ac2e7b3b Accepted     D31941: Use a char * to avoid alignment warnings.
b2d94deacfa1 Needs Review D31945: libmd: Only define SHA256_Transform_c when using the ARM64 ifunc.
c9be458cee94 Accepted     D31948: mana: Cast an unused value to void to quiet a warning.
8a9b7debfc0c No Review    : arm64: Add compat macros for system registers for GNU as.
```

Updating Reviews

Using the update subcommand with a branch is not quite as straightforward as the other commands. In particular, the update subcommand is not able to determine if the review associated with a commit is already up to date (and thus should not be updated). Instead, it will always update all reviews if it is given the list of reviews for a branch. The update subcommand also prompts for a description for each commit. With a branch, it is generally better to use the update subcommand for individual commits after they have been changed rather than running the subcommand against an individual branch.

Finalizing the Reviews and Pushing the Branch

Once all the commits are ready for pushing, the stage subcommand can be used to stage all of the commits in the branch in a single operation. The user's editor will be invoked to finalize

the commit log of each commit. Once the operation completes, the branch can be pushed via `git push` as described earlier in the single commit example.

Individual commits in a branch can also be pushed by using the stage subcommand with specific commits. After pushing those commits, rebase the branch to remove the pushed commits from the branch. The `list` subcommand can then be used to examine the status of the remaining commits on the branch.

Limitations and Caveats

While git-arc provides a streamlined interface on top of **arc**, it has several limitations of its own:

- Matching a commit with an existing review requires the first line of the commit log to match the title of the review. This means that if you update the first line of the commit log for a commit with an existing review, the review title must be manually updated in Phabricator before the update, `list`, or `stage` subcommands will recognize the review.
- Similarly, the update subcommand is only able to update the patch for a given review. It does not update the review description if the commit log message has changed.
- If commits are dropped from a branch while it is in review, git-arc does not provide a way to notice that commits are dropped or to abandon the associated reviews. Abandoning the reviews must be done in Phabricator instead.
- If additional commits are added to a branch while it is in review, the list subcommand can be used to find those commits and the create subcommand can be used to create reviews. However, git-arc does not provide a way to auto-adjust the parent / child relationships in Phabricator to update the stack to match the new layout of the branch. This must be done manually in Phabricator instead.
- git-arc is not able to determine if a commit's review is stale (that is, if a commit has been updated since the review was created or last updated). Such functionality would be useful to annotate stale revisions in the `list` subcommand. It could also make the update subcommand more useful with revision ranges by omitting updates to unchanged commits.

JOHN BALDWIN is a systems software developer. He has directly committed changes to the FreeBSD operating system for 20 years across various parts of the kernel (including x86 platform support, SMP, various device drivers, and the virtual memory subsystem) and userspace programs. In addition to writing code, John has served on the FreeBSD core and release engineering teams. He has also contributed to the GDB debugger and LLVM. John lives in Concord, California, with his wife, Kimberly, and three children: Janelle, Evan, and Bella.

How to Implement a Simple USB Driver for FreeBSD

BY MARIUSZ ZABORSKI

Have you ever bought new equipment for your FreeBSD box and it turned out that some functionality didn't work? Instead of returning it, it may be possible to write your own driver without much effort. We will explain how to write a simple USB driver for FreeBSD.

Case Study

In this article, we will look into a driver for Razer Ornata V2. The device is a Mecha-membrane keyboard which works perfectly on FreeBSD with one small issue: you can't change the backlight color. In some cases, you may find a keyboard that has a built-in color change. This means that the color will change independently on the software run on your machine under some key combination. In the case of this keyboard, the driver in operating systems controls the backlight. Thanks to that, you can have fancy patterns on your keyboard like fire flames. The disadvantage is that you have to have a driver for it. The device is shown in Figure 1.

Figure 1. Razer Ornata V2

Gathering the Information

First of all, we have to understand the protocol used in the driver. In the case of drivers for Razer, we have two ways of doing it:

- Look into an openrazer (unofficial collection of Linux drivers for Razer devices)
- Sniff the USB protocol from the Windows driver

In this article, we will combine these two methods. When we initially looked into the problem, there wasn't support for Razer Ornata V2 in the openrazer, so we had to deduct some of the parts from a USB protocol dump. The support for this keyboard was recently added to the openrazer, but when you try to write your driver, parts of it may not be available anywhere else than in the official Windows drivers. For educational purposes, we will assume that the openrazer doesn't support this keyboard.

OpenRazer

To get a needed context about the driver, we will try to find the package structure used to communicate with the keyboard, as this allows us to understand the dump from the USB sniff. The source code for openrazer is available under https://github.com/openrazer/openrazer. In a driver/razercommon.h file, we will find a `razer_report` structure, which is the main structure of the driver. It is used across all of the devices from this product. The structure is shown in Listing 1.

Listing 1. The razer_report structure defined by openrazer

```
struct razer_report {
    unsigned char status;
    union transaction_id_union transaction_id; /* */
    unsigned short remaining_packets; /* Big Endian */
    unsigned char protocol_type; /*0x0*/
    unsigned char data_size;
    unsigned char command_class;
    union command_id_union command_id;
    unsigned char arguments[80];
    unsigned char crc;/*xor'ed bytes of report*/
    unsigned char reserved; /*0x0*/
};
```

Sniffing a Windows Driver

To sniff a Windows USB driver, we can use a usbpcap (https://desowin.org/usbpcap/) tool. It is a command-line tool that is very simple to use (in Listing 2, we have an example). When we run the command tool, it will show us available devices; next, it will ask us which device we want to sniff and where to save a pcap file. The generated pcap file is easily viewable using Wireshark.

We will be targeting a Razer Windows driver. On Windows, the Razer Synapse tool allows you to customize the backlight colors of the keyboard. Let's try to set up different colors of the keyboard while the usbpcap is running. Thanks to this tool, we will record all requests sent to the keyboard (the Razer Synapse is shown in Figure 2). At this point, we will apply the red scheme on the whole keyboard.

Listing 2. Usage of usbpcap to capture the USB protocol

```
Following filter control devices are available:
1 \\.\USBPcap1
  \??\USB#ROOT_HUB20#4&19d0fd2a&0#{f18a0e88-c30c-11d0-8815-00a0c906bed8}
    [Port 1] Generic USB Hub
      [Port 4] ThinkPad Bluetooth 4.0
      [Port 6] Integrated Camera

2 \\.\USBPcap2
\??\USB#ROOT_HUB20#4&182122df&0#{f18a0e88-c30c-11d0-8815-00a0c906bed8}
    [Port 1] Generic USB Hub

3 \\.\USBPcap3
\??\USB#ROOT_HUB30#4&23ace5cb&0&0#{f18a0e88-c30c-11d0-8815-00a0c906bed8}
    [Port 1] Generic USB Hub
      Razer Ornata V2
        Razer Ornata V2
           Razer Ornata V2
        Razer Ornata V2
          Razer Control Device
Select filter to monitor (q to quit): 3
Output file name (.pcap): t1.pcap
```

Combining Methods

Now that we have a pcap from the dump, we can start analyzing the recorded protocol. Don't get into too much detail on how USB drivers work; instead, glean the general idea about the protocol. Most of the values we will just copy, as we might need to change them. We only want to generate similar requests as the original driver.

In Figure 3, we can see a dump created using usbpcap under Wireshark; in this case, the driver uses a setup packet. The setup packets are used for detection and configuration of the USB devices. In Table 1, we can see a package defined by the USB specification as well as the values that were sent by the driver.

Figure 2. Razer Synapse tool. The tool is used to configure the backlight color.

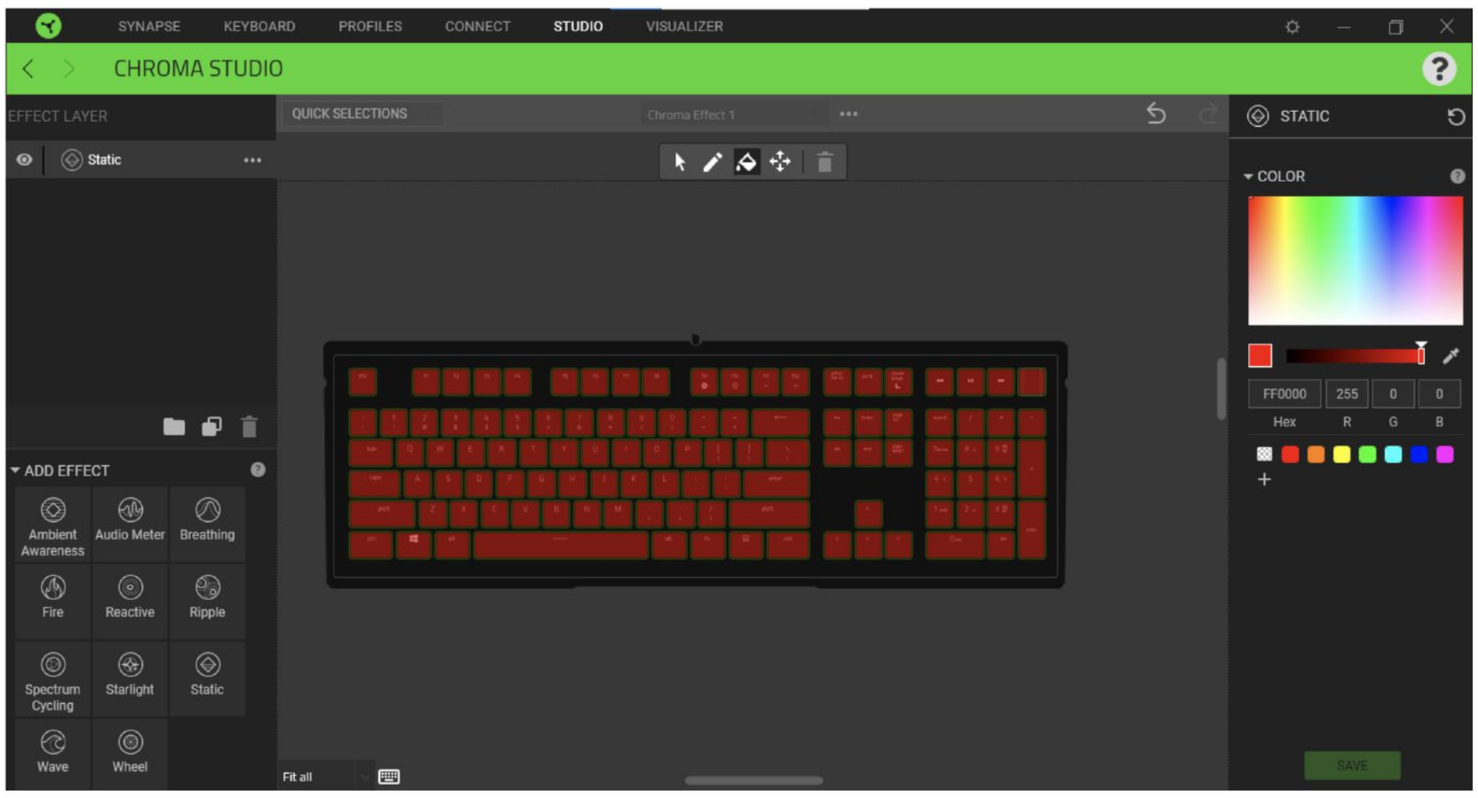


Table 1. Format of Setup Data from USB documentation, with the values from the dump.

Offset	Field	Size	Value	Description	Values from pcap
0	*bmRequestType*	1	Bitmap	Characteristics of request: • D7: Data transfer direction 0 = Host-to-device 1 = Device-to-host • D6...5:Type 0 = Standard 1 = Class 2 = Vendor 3 = Reserved • D4...0:Recipient 0 = Device 1 = Interface 2 = Endpoint 3 = Other • 4...31 = Reserved	0x21 • Data transfer direction: *Host-to-device* • Type: *Class* • Recipient: *Interface*
1	*bRequest*	1	Value	Specific request (for more details please refer to the USB Specification)	0x09 SET_REPORT
2	*wValue*	2	Value	Word-sized field that varies according to request	0x300
4	*wIndex*	2	Value or Offset	Word-sized field that varies according to request; typically used to pass an index or offset	2
6	*wLength*	2	Count	Number of bytes to transfer if there is a data stage	90

After the setup data, we have specific data for a Razer Driver. In Figure 4, we combined the pcap data with the `razer_report` structure from the openrazer project. Next, we can easily see some more things about the arguments. First, we have 2 bytes set to 0, which we can assume are reserved. Next, we have a one byte set to 1. When we look into the pcap, we can see many similar packages that, in this place, have this value in range from 0 to 5. We can verify this later, but actually it seems that this is the row number on the keyboard. Then, we have a value 0x15 (21), which is actually the number of keys in a row. Finally, there is a 21-times repeated value 0xff0000, which seems to refer to the red color that we set in RGB (R: 255, G:0, B:0).

Figure 3. The pcap generated using usbpcap under Wireshark. The Setup packet is highlighted.

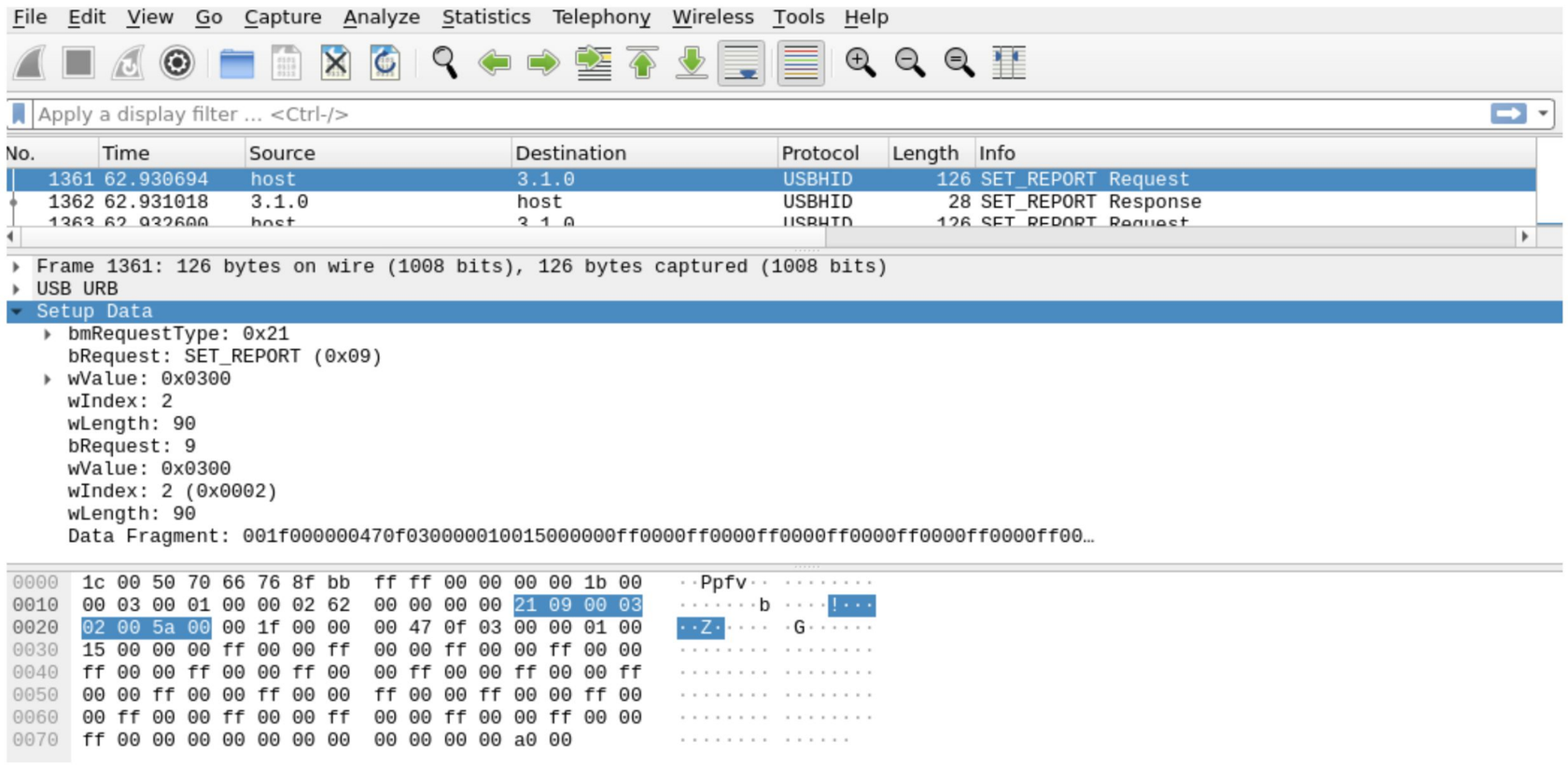


Figure 4. The setup data with the structure obtained from openrazer.

```
0000 00 1f 00 00 00 47 0f 03 00 00 01 00 15 00 00 00
0010 ff 00 00 ff 00 00 ff 00 00 ff 00 00 ff 00 00 ff
0020 00 00 ff 00 00 ff 00 00 ff 00 00 ff 00 00 ff 00
0030 00 ff 00 00 ff 00 00 ff 00 00 ff 00 00 ff 00 00
0040 ff 00 00 ff 00 00 ff 00 00 ff 00 00 ff 00 00 00
0050 00 00 00 00 00 00 00 00 a0 00
```

Fields	Values
Status	0x00
Transaction ID	0x1f
Remaining packets	0x0000
Protocol Type	0x00
Data Size	0x47
Command class	0x0f
Command ID	0x03
Arguments	0000010015
CRC	0xa0
Reserved	0x00

The packages used by the Razer Synapse seem to allow us to set each key to a different color. Each package refers to a single row on the keyboard. Without going into too much detail of USB and Razer protocol, this should be enough to implement any type of backlight effect we might like.

Lastly, we need to find a vendor and product identifier which will allow us to find the correct USB device. To do that, we can use the usbconfig(8) tool on the FreeBSD box. The utility has a special option, dump_device_desc, which allows us to print the details of all USB devices connected to our box. The example of usage is shown in Listing 3.

Listing 3. Identifying vendor and product ID using usbconfig(8) tool.

```
# usbconfig dump_device_desc
ugen0.4: <Razer Razer Ornata V2> at usbus0, cfg=0 md=HOST spd=FULL (12Mbps)
pwr=ON (500mA)

  bLength = 0x0012
  bDescriptorType = 0x0001
  bcdUSB = 0x0200
  bDeviceClass = 0x0000  <Probed by interface class>
  bDeviceSubClass = 0x0000
  bDeviceProtocol = 0x0000
  bMaxPacketSize0 = 0x0040
  idVendor = 0x1532
  idProduct = 0x025d
  bcdDevice = 0x0200
  iManufacturer = 0x0001  <Razer>
  iProduct = 0x0002  <Razer Ornata V2>
  iSerialNumber = 0x0000  <no string>
  bNumConfigurations = 0x0001
```

libusb&PyUSB

The first way of implementing a simple driver is to use libusb and PyUSB. This method allows us to write a USB driver in a userland without any additional kernel modules. Writing drivers in a userland is the most secure, because if there is a bug, it will expose only the kernel part for attack.

The libusb is a library for USB devices. It is a cross platform library, so we can see a port of it in FreeBSD, Linux, OpenBSD or even Windows. To simplify the task even more, instead of writing a driver in C, we can implement it using Python, which is possible thanks to the PyUSB module. PyUSB provides easy access to a host USB system. We can simply install pyusb using the pkg(8) tool (e.g. `pkg install py38-pyusb`).

First, we have to find a valid device. To do that, we use a usb.core.find function. To identify the right device, we can provide a product and vendor ID obtained from usbconfig(8), which is shown in Listing 4.

Listing 4. Finding a device using PyUSB.

```
# python
Python 3.8.10 (default, Jul  6 2021, 01:34:57)
>>> import usb.core
>>> dev = usb.core.find(idVendor=0x1532, idProduct=0x025d)
>>> dev.product
'Razer Ornata V2'
```

To send a Setup packet, we use the `ctrl_transfer` function. The interface of this function corresponds to the parameters described in the Setup packet. The simplest thing to do here is to copy all sniffed parameters. The last step is to rebuild the package. In our driver, we will assume that the color is hardcoded. Besides the color, row and CRC field, we will copy all of them from the sniffed part (the whole process is shown in Listing 5). At the end, we also have to recalculate the CRC field.

Listing 5. Sending a request to change a color using PyUSB.

```
import usb.core

# Color
r = 0xff
g = 0x00
b = 0x00

def change_color(dev, line, r, g, b):
    # Recreate package
    package = bytes([
        0x00,           # Status
        0x1f,           # Transaction ID
        0x00, 0x00,     # Remaining packets
        0x00,           # Protocol Type
        0x47,           # Data Size
        0x0f,           # Command Class
        0x03,           # Command ID
```

```
                # Arguments:
        0x00,   # - unknown
        0x00,   # - unknown
        line,   # - line
        0x00,   # - unknown
        0x15,   # - number of keys
        0x00, 0x00, # - unknown
        0x00    # - unknown
])
for _ in range(0x15):
    package += bytes([r, g, b])

# Fill up to 0x47 bytes size
for _ in range(0x3):
    package += bytes([0, 0, 0])

# Recalculate crc
crc = 0x00
for x in package:
    crc ^= x
package += bytes([crc, 0x00]) # crc and reserved
dev.ctrl_transfer(
    bmRequestType = 0x21,
    bRequest = 0x09,
    wValue = 0x300,
    wIndex = 0x02,
    data_or_wLength = package
)

dev = usb.core.find(idVendor=0x1532, idProduct=0x025d)
for line in range(6):
    change_color(dev, line, r, g, b)
```

Kernel Module

In the case of a native driver, we have to write a FreeBSD kernel module. We also have to implement some kind of communication between the kernel and the userland to tell the module what color we want. To accomplish this, we can expose some additional sysctl, implement a iocotl(9) or read the input from the USB dev node. In this article, we will look at the ioctl(9) method.

Building a Kernel Module

First, we have to know how to compile the kernel module. The simplest way of doing this is using a Makefile and including the bsd.kmod.mk file. Thanks to that, it will auto generate all additional required files and headers. We also have to remember to include files like opt_usb.h, buf_if.h and device_if.h., which is common for all kernel modules. In the KMOD detective, we provide the name of the compiled driver. The example of Makefile is shown in Listing 6.

Listing 6. Makefile for building kernel module in FreeBSD.

```
SRCS=ornata.c
SRCS+=opt_usb.h bus_if.h device_if.h

KMOD=ornata

.include <bsd.kmod.mk>
```

The three standard methods that almost all drivers have to implement is probe, attach and detach. There are also additional methods, like suspend and resume, but we won't look into them.

The probe is executed first to examine the device and decide if the driver is supported or not. Here, we can use a VendorID and ProductID to decide if this is the device we are looking for. We can accomplish that using a `usbd_lookup_id_by_uaa` function, which will iterate over the given array of vendors and products to find a matching pair. We also have to check if the device is in host mode (`USB_MODE_HOST`), which is needed to initiate data transfers. Next, we want to be sure the device is actually a keyboard. The probe function is shown in Listing 7.

Listing 7. The USB probe function

```
static const STRUCT_USB_HOST_ID ornata_devs[] = {
	{USB_VPI(0x1532, 0x025d, 0)},
};

static int
ornata_probe(device_t self)
{
	struct usb_attach_arg *uaa = device_get_ivars(self);

	if (uaa->usb_mode != USB_MODE_HOST)
		return (ENXIO);

	if (uaa->info.bInterfaceProtocol == UIPROTO_BOOT_KEYBOARD)
		return (ENXIO);

	if (uaa->info.bInterfaceClass != UICLASS_HID)
		return (ENXIO);

	return (usbd_lookup_id_by_uaa(ornata_devs, sizeof(ornata_devs), uaa));
}
```

Two other methods that are useful are attach and detach. The attach function is called when the probe phase is finished and the probe function returns success. It is an entry point that allows the driver to initialize all required resources. At the opposite side, we have a detach function that allows us to clean up after the device disappears.

In case of this, the driver in the attached function will initialize mutex needed for synchro-

nizing and allocate the USB driver's entry points under /dev. The last part is done by the `usb_fifo_attach` function. While creating a new node, we have to also define what operations it supports (the `ornata_fifo_methods` variable), but we will look into that in the next phases. While creating a node, we can define which user and group should be an owner (in our case root(0) and wheel(0) group) and in what mode the node should be initialized (in our case everyone can read and write (666)). At this moment, we also introduce a helping structure which stores all device specific variables. At the opposite side, in the detach routine, we call the `usb_fifo_detach` function, which destroys its associated USB device node. These functions are shown in Listing 8.

Listing 8. Attach and detach function for the driver.

```
struct ornata_softc {
        struct usb_fifo_sc sc_fifo;
        struct mtx sc_mtx;

        struct usb_device  *sc_udev;
};

static int
ornata_attach(device_t self)
{
        struct usb_attach_arg *uaa = device_get_ivars(self);
        struct ornata_softc *sc = device_get_softc(self);
        int unit = device_get_unit(self);
        int error;

        device_set_usb_desc(self);
        mtx_init(&sc->sc_mtx, "ornata lock", NULL, MTX_DEF);

        error = usb_fifo_attach(uaa->device, sc, &sc->sc_mtx,
            &ornata_fifo_methods, &sc->sc_fifo,
            unit, -1, uaa->info.bIfaceIndex,
            0, 0, 0666);
        if (error)
                goto detach;

        sc->sc_udev = uaa->device;

        return (0);
detach:
        mtx_destroy(&sc->sc_mtx);
        return (error);
}

static int
ornata_detach(device_t self)
{
        struct ornata_softc *sc = device_get_softc(self);
```

```
        usb_fifo_detach(&sc->sc_fifo);
        mtx_destroy(&sc->sc_mtx);

        return (0);
}
```

Finally, we can define the driver module, which is shown in Listing 9. We are creating a kernel driver using a DRIVER_MODULE macro. In this part, we are setting the probe, attach and detach function to the strcuture driver. The MODULE_DEPEND macro is used to set the dependency on another kernel module. This is only used to help the operating system to load all required modules before loading this one; however, this does not dictate the order of the load.

Listing 9. Definition of kernel module.

```
static device_method_t ornata_methods[] = {
        DEVMETHOD(device_probe, ornata_probe),
        DEVMETHOD(device_attach, ornata_attach),
        DEVMETHOD(device_detach, ornata_detach),

         DEVMETHOD_END
};

static driver_t ornata_driver = {
        .name   = "ornata",
        .methods = ornata_methods,
        .size = sizeof(struct ornata_softc)
};

static devclass_t ornata_devclass;

DRIVER_MODULE(ornata, uhub, ornata_driver, ornata_devclass, NULL, 0);
MODULE_DEPEND(ornata, ukbd, 1, 1, 1);
MODULE_VERSION(ornata, 1);
USB_PNP_HOST_INFO(ornata_devs);
```

At this point, we can implement a function that will send setup data to the device. This can be done using the `usbd_do_request_flags` function and the `usb_device_request` structure representing the request. For the data part, we can use the structure from openrazer, as it is already implemented in the C language. For example, in the case of the python driver, the function will expect the color and the line to set, and most of the variables are just copied from our sniffed requests. We also have to remember to recalculate the CRC field. The USET macros allow us to set data independent of CPU endianness. The function for setting the backlight color is shown in Listing 10.

Listing 10. Attach and detach function for the driver.

```
static void
ornata_set_color(struct ornata_softc *sc, uint8_t r, uint8_t g, uint8_t b, uint8_t
line)
{
        struct razer_report rr;
        struct usb_device_request req;
```

```
char crc, *ptr;
int i;

memset(&rr, 0, sizeof(rr));

req.bmRequestType = 0x21;
req.bRequest = 0x09;
USETW(req.wValue, 0x300);
USETW(req.wIndex, 2);
USETW(req.wLength, sizeof(rr));

rr.status = 0x00;
rr.transaction_id = 0x1f;
rr.remaining_packets = 0x00;
rr.protocol_type = 0x00;
rr.data_size = 0x47;
rr.command_class = 0x0f;
rr.command_id = 0x03;

rr.arguments[2] = line;
rr.arguments[4] = 0x15;

for (i = 8; i < 8 + 0x15 * 3; i += 3) {
        rr.arguments[i] = r;
        rr.arguments[i + 1] = g;
        rr.arguments[i + 2] = b;
}

crc = 0;
for (ptr = (char *)&rr; ptr != (char *)&rr + sizeof(rr); ptr++) {
        crc ^= *ptr;
}

rr.crc = crc;

usbd_do_request_flags(sc->sc_udev, &sc->sc_mtx, &req,
    &rr, 0, NULL, 2000);
}
```

Implementing ioctl

The only missing part in the driver is the methods used to communicate with the USB device node. We will implement the ioctl method, as it is the simplest (but requires an additional program to send an ioctl).

First, we have to define the ioctl. To accomplish this, we can use _IOW macro, which defines a macro for a write operation — which means that the memory will be copied from userland to the kernel. For other purposes, we can use _IOR to define a read ioctl, or _IOWR for read/write operation, or _IO, which transfers no data. We will also use an additional structure, `ornata_color`, just to transfer the data in an organized way.

The definition of ioctl is shared between the userland and the kernel, so a good idea is to define a C file header that contains these definitions. The header is shown in Listing 11.

Listing 11. Attach and detach function for the driver.

```
#ifndef _ORNATA_H_
#define _ORNATA_H_

#include <sys/ioccom.h>

struct ornata_color {
        uint8_t r;
        uint8_t g;
        uint8_t b;
};

#defineORNATA_SET_COLOR _IOW('U', 205, struct ornata_color)

#endif
```

Now, getting back to the `usb_fifo_attach`, we use a structure `ornata_fifo_methods` that hasn't yet been defined. This structure defines supported operations on the device; for example, open or close. In our case, we want to support ioctl operations. The basename field describes the name of the node that should be created under /dev. When using the ioctl, the memory is already safely copied from the userland to the kernel, so we can just use color structure. The implementation of ioctl is shown in Listing 12.

Listing 12. Implementation of ioctl method.

```
static int
ornata_ioctl(struct usb_fifo *fifo, u_long cmd, void *addr, int fflags)
{
        struct ornata_softc *sc;
        struct ornata_color color;
        int error;
        uint8_t line;

        sc = usb_fifo_softc(fifo);
        error = 0;

        mtx_lock(&sc->sc_mtx);

        switch(cmd) {
        case ORNATA_SET_COLOR:
                color = *(struct ornata_color *)addr;
                for (line = 0; line < 6; line ++) {
                        ornata_set_color(sc,
                          color.r,
                          color.g,
                          color.b,
                          line);
                }
```

```
                break;
        default:
                error = ENOTTY;
                break;
        }

        mtx_unlock(&sc->sc_mtx);
        return (error);
}

static struct usb_fifo_methods ornata_fifo_methods = {
        .f_ioctl = &ornata_ioctl,
        .basename[0] = "ornata"
};
```

The disadvantage of this approach is that we have to implement an additional userland program, because there is no way of generating the ioctl(2) from a command line. This program is shown in Listing 13.

Listing 13. Example of usage of ioctl in userland.

```
int
main(void)
{
        int fd = open("/dev/ornata0", 0);
        struct ornata_color color;

        color.r = 0xFF;
        color.g = 0x00;
        color.b = 0x00;

        ioctl(fd, ORNATA_SET_COLOR, &color);

        return (0);
}
```

Summary

Implementing a userland driver isn't that complicated, thanks to libusb and pyusb. The most complicated part is actually understanding the protocol used by the device. If the protocol is simple, we can just sniff a lot of data from existing drivers on different platforms. If the protocol is more complicated, maybe there is an open-source project and we can port some part of it to FreeBSD. In the case of writing a native driver, we have to be patient, as the routines are more challenging. Implementing the kernel driver, we have to be very careful, as we can introduce bugs. Also, if we mess up something, the kernel may just panic, and we will need to restart the machine.

Bibliography

- USB 2.0 Specification — https://www.usb.org/document-library/usb-20-specification
- *FreeBSD Device Drivers A Guide for the Intrepid* by Joseph Kong
- Openrazer source code — https://github.com/openrazer/openrazer
- Roland's homepage — Setting the Razer ornata chroma color from userspace (https://rsmith.home.xs4all.nl/hardware/setting-the-razer-ornata-chroma-color-from-userspace.html)

MARIUSZ ZABORSKI currently works as a security expert at 4Prime. Since 2015, he has been the proud owner of the FreeBSD commit bit. His main areas of interest are OS security and low-level programming. In the past, he worked at Fudo Security, where he led a team developing the most advanced PAM solution in IT infrastructure. In 2018, Mariusz organized the Polish BSD User Group. In his free time, he enjoys blogging at https://oshogbo.vexillium.org.

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!

2021 Editorial Calendar

- Case Studies (January-February)
- FreeBSD 13 (March-April)
- Security (May-June)
- Desktop/Wireless/Graphics (July-August)
- FreeBSD Development (September-October)
- Storage (November-December)

Find out more at: freebsdfoundation/journal

Kernel Development Recipes

BY MARK JOHNSTON

A question that I frequently see is something along the lines of, "How do I get started in operating system kernel development?" This is difficult to answer in general, but there is a simpler side to the question, which is, "How can I set myself up to (efficiently) build and test kernel changes?" In other words, while submitting one's first kernel patch is a significant milestone, a frequent contributor may be working on multiple different patches, testing patches (possibly obtained from other developers), bisecting to find the cause of a regression, or debugging a kernel panic. It is important to have a workflow that minimizes the amount of time spent waiting or fiddling with configuration knobs or shell scripts.

FreeBSD, being a large and rather long-lived project (even older than some of its developers) and targeting a variety of hardware platforms (from palm-sized SoCs to large multi-core servers with terabytes of RAM), does not have a one-size-fits-all approach to these kinds of tasks. However, the benefits of a fast edit-compile-test loop are universal; this article seeks to illustrate a few tricks which can reduce the friction of many kernel development tasks.

The steps below assume a FreeBSD host and POSIX-compatible shell, such `as /bin/sh`. While the project currently does not provide much in the way of canned scripts for building, booting and testing kernel changes, some of the suggestions here can be incorporated into one's development environment or into a CI system shared by multiple developers.

Git Worktrees

Before starting any kind of work on FreeBSD one needs a copy of the src repository:

```
$ git clone https://git.freebsd.org/src.git freebsd
```

When working on multiple tasks, it will be useful to have multiple copies of the FreeBSD source files. Having a single copy becomes awkward if a build of the currently checked out branch is running and while waiting you want to switch to a different branch to work on something else, or possibly because switching branches would update timestamps on source files and thus unnecessarily slow down future incremental builds.

It is of course possible to keep multiple clones of a repository, but a better solution is to use `git` worktrees, which let one checkout multiple branches from a single clone. This requires less

disk space and ensures that all of your work is contained in a single copy of the repository. For example, it can be useful to have worktrees for frequently accessed stable and release branches:

```
$ cd freebsd
$ git worktree add dev/stable/13 origin/stable/13
$ git worktree add dev/releng/13.0 origin/releng/13.0
$ git worktree add dev/stable/12 origin/stable/12
$ git worktree add dev/releng/12.2 origin/releng/12.2
...
```

Note that worktrees may be located outside of the clone, for example:

```
$ git worktree add ../freebsd-stable/13 origin/stable/13
```

When working on a larger project, it is useful to keep a worktree dedicated to that branch.

Building and Booting a Custom Kernel — Quickly

Suppose that you have written a small, simple kernel patch and want to do some sanity testing before submitting it for review. The standard command for building a FreeBSD kernel is well-known:

```
$ cd freebsd
$ make buildkernel
```

This will perform a clean, single-threaded kernel build. It is usually preferable to use as much CPU as is available, so consider adding `-j $(sysctl -n hw.ncpu)` to the `make(1)` flags: this will run as many build jobs in parallel as there are CPUs in the system. Once a kernel has been built from a particular source path, it is not usually necessary to rebuild every single source file each time a change needs to be tested — an incremental rebuild is sufficient. To request an incremental build, add the `-DKERNFAST` flag to the `make(1)` invocation:

```
$ make -j $(sysctl -n hw.ncpu) -DKERNFAST buildkernel
```

By default, a `buildkernel` will rebuild not only the kernel, but also every single kernel module that comes with it, which on an amd64 system running a `GENERIC` kernel comes out to 822 modules. Many of these modules are also linked into the kernel proper (depending on the kernel configuration file in use), so a `buildkernel` may spend considerable time rebuilding modules that will never be loaded. The `MODULES_OVERRIDE` variable can be used to override this behaviour; instead of building a kernel and all modules, a `buildkernel` with `MODULES_OVERRIDE` will build only a kernel and the modules specified by the variable's value. For example, to build a kernel together with only the `tmpfs.ko` and `nullfs.ko` modules, run the following:

```
$ make -j $(sysctl -n hw.ncpu) -DKERNFAST \
  MODULES_OVERRIDE="tmpfs nullfs" buildkernel
```

If you know that your testing requires only a handful of modules, your build times can benefit from setting `MODULES_OVERRIDE` since this may greatly reduce the number of filesystem accesses performed during a build. If the source tree is located on a slow disk, or remotely and accessed over NFS, for example, the `MODULES_OVERRIDE` option may save a lot of time.

Now that a test kernel is available, it can be tested. The procedure for testing a change depends heavily on the nature of the change; a developer working on a WiFi driver will have different testing workflows than a developer working on bringing up FreeBSD on a new platform or on the kernel memory allocator. For many purposes, however, a simple `bhyve` virtual machine (VM) is sufficient.

The FreeBSD project provides VM images that are handy for use in testing kernel changes. While it is of course possible to build VM images locally, using the `release(7)` scripts for example, the pre-built images are quite convenient:

```
$ IMAGE="FreeBSD-14.0-CURRENT-amd64.raw.xz"
$ URL="https://ftp.freebsd.org/pub/FreeBSD/snapshots/VM-IMAGES/14.0-CURRENT/amd64/Latest"
$ fetch -o "/tmp/${IMAGE}" "${URL}/${IMAGE}"
$ unxz "/tmp/${IMAGE}"
$ sudo pkg install -y bhyve-firmware
$ sudo sh /usr/share/examples/bhyve/vmrun.sh -E -d \
  "/tmp/${IMAGE%.xz}" myvm
```

This boots up the snapshot image using bhyve, in a VM called “myvm”. However, the VM will be running the kernel included with the snapshot. There are several ways to update the image’s kernel. For instance, a copy of the source tree could be copied or mounted into the booted VM, and a new kernel can be built within the VM. However, this will be slow unless the VM is given a large amount of CPU and RAM, and poses the logistical problem of exporting the source tree. Another approach could be to mount the disk image on the host and install a new kernel directly, but this requires some synchronization to ensure that the VM and the host do not have the image mounted at the same time.

A different approach is to provide a second disk containing only the custom kernel and modules, and configuring the VM to boot from that instead. Such disk images can be created quickly and without any special privileges on the host. First, the following commands can be used to create such an image:

```
$ cd /usr/src
$ make buildkernel
$ make installkernel -DNO_ROOT DESTDIR=/tmp/kernel
$ cd /tmp/kernel
$ makefs -B little -S 512 -Z -o version=2 /tmp/kernfs METALOG
$ rm -f /tmp/kernfs.raw
$ mkimg -s gpt -f raw -S 512 -p freebsd-ufs/kern:=/tmp/kernfs \
  -o /tmp/kernfs.raw
```

The `make` commands build and install a kernel to `/tmp/kernel` without requiring root privileges. This also creates an `mtree(8)` manifest in `/tmp/kernel/METALOG` which is used by

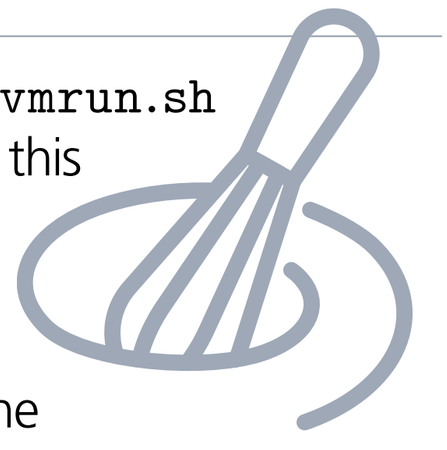
`makefs(8)` to build a small filesystem. Finally, `mkimg(1)` adds a GPT to the filesystem, making it accessible to the FreeBSD boot loader.

Now we can boot the VM again with the extra disk:

```
$ sudo sh /usr/share/examples/bhyve/vmrun.sh -E -d /tmp/kernfs.raw \
  -d "${IMAGE%.xz}" myvm
```

This still boots the kernel from the original image. However, the boot loader can be configured to load the kernel from `kernfs.raw` instead, by adding the following line to `/boot/loader.conf`:

```
kernel="disk0p1:/boot/kernel"
```



"`disk0p1`" here refers to the first partition of disk0, the first disk listed in the `vmrun.sh` command-line arguments, which in this case is `/tmp/kernfs.raw`. After making this change and rebooting, the VM should boot into the custom kernel.

The custom kernel filesystem is not mounted by default, which means that tools like `kldload(8)` will not be able to automatically load kernel modules corresponding to the custom kernel. To remedy this, add the following lines to the corresponding system configuration files:

`/boot/loader.conf`:

```
# Make sure that nullfs is available.
nullfs_load="YES"
```

`/etc/sysctl.conf`:

```
# Fix up paths used by the kernel linker.
kern.bootfile=/boot/kernel/kernel
kern.module_path=/boot/kernel
```

`/etc/fstab`:

```
# Mount the custom kernel filesystem at /boot/kernel.
/dev/gpt/kern /mnt ufs ro 0 0
/mnt/boot/kernel /boot/kernel nullfs ro 0 0
```

Finally, consider adding `autoboot_delay=1` to `/boot/loader.conf`: this reduces the loader delay from ten seconds to one, which helps considerably when reboots are frequent.

While the setup was a bit involved, it only needs to be performed once, and we now have a way to quickly boot up a freshly built kernel! Kernel builds can be done while the VM is running, and rebooting the VM will cause the latest kernel build to be loaded and run. When creating more elaborate testing environments, it may be desirable to build the base VM image locally as well, in which case the configuration updates described above may be automated.

Debugging a Custom Kernel

One benefit of using `bhyve(8)` is the GDB protocol stub available to the host. QEMU has a similar feature. Using this, the host system can run a debugger on the guest kernel. Since our custom kernel was built on the host, this functionality is trivial to use. The `vmrun.sh` currently does not support enabling the GDB stub, but it can be enabled using a raw bhvye(8) invocation:

```
# bhyve -c 1 -m 512M -H -A -P -G :1234 \
        -s 0:0,hostbridge \
        -s 1:0,lpc \
        -s 2:0,virtio-blk,kernfs.raw \
        -s 3:0,virtio-blk,FreeBSD-14.0-CUREENT-amd64.raw \
        -l com1,stdio \
        -l bootrom,/usr/local/share/uefi-firmware/BHYVE_UEFI.fd \
        myvm
```

Here, the `-G :1234` parameter instructs `bhyve(8)` to listen on port 1234 for connections from a debugger. When booting a VM, `bhyve(8)` may optionally pause waiting for a connection before booting the kernel; this is handy for debugging problems that arise early during kernel boot-up. To enable this, specify `-G w:1234`.

While the VM is running (or waiting for a connection), the `kgdb` program (installed with the `gdb` package) can attach to the guest using port 1234:

```
$ kgdb -q /tmp/kernel/usr/lib/debug/boot/kernel/kernel.debug
Reading symbols from /tmp/kernel/usr/lib/debug/boot/kernel/kernel.debug...
(kgdb) target remote localhost:1234
Remote debugging using localhost:1234
cpu_idle_acpi (sbt=432162053) at
/usr/home/markj/src/freebsd/sys/x86/x86/cpu_machdep.c:551
551             atomic_store_int(state, STATE_RUNNING);
(kgdb) set solib-search-path /tmp/kernel/usr/lib/debug/boot/kernel
Reading symbols from
/tmp/kernel/usr/lib/debug/boot/kernel/nullfs.ko.debug...
(kgdb) bt
#0  cpu_idle_acpi (sbt=432162053) at
/usr/home/markj/src/freebsd/sys/x86/x86/cpu_machdep.c:551
#1  0xffffffff81096ccf in cpu_idle (busy=0) at
/usr/home/markj/src/freebsd/sys/x86/x86/cpu_machdep.c:668
#2  0xffffffff80c62b41 in sched_idletd (dummy=<optimized out>,
dummy@entry=0x0 <nullfs_init>)
    at /usr/home/markj/src/freebsd/sys/kern/sched_ule.c:2952
#3  0xffffffff80be838a in fork_exit (callout=0xffffffff80c62660
<sched_idletd>, arg=0x0 <nullfs_init>,
    frame=0xfffffe0001141f40) at
/usr/home/markj/src/freebsd/sys/kern/kern_fork.c:1088
#4  <signal handler called>
(kgdb)
```

At this point the VM is suspended waiting for a command from the debugger. The `continue` command can be used to resume the VM, and hitting ctrl-C in the debugger window will suspend

the VM again. This functionality is extremely useful for debugging kernel deadlocks and boot-time problems. It is also possible to attach a debugger after the guest kernel has panicked, making it easy to inspect threads and local variables without having to configure and recover kernel dumps.

Testing a Custom Kernel

We now have a fairly quick loop for incrementally testing changes to the FreeBSD kernel, as well as some tooling for debugging problems when they arise. At this point some manual testing might indicate that the patch is correct and ready for review. It may be useful though to run some automated tests to increase confidence in the patch; the FreeBSD kernel is a large, monolithic body of code, and there is always some potential for unforeseen regressions. A good place to start is the FreeBSD regression test suite.

If you followed the steps above to set up a kernel development VM, then the test suite is already installed; use the following commands to run it:

```
# cd /usr/tests
# kyua -v test_suites.FreeBSD.allow_sysctl_side_effects=1 test
```

The `allow_sysctl_side_effects` flag enables tests which depend on being able to modify global sysctl values, which is perfectly fine in a dedicated VM. Some tests will also be skipped if they depend on third-party ports, such as Python. After a run, a summary of results (including skipped tests) can be viewed with:

```
# kyua report
```

A `bhyve` VM can be set up to automatically run the test suite upon booting up. One simple way to achieve this is to add an `/etc/rc.local` script which runs the test suite, prints results to the console, and shuts down the VM. A separate disk could be used to store the output of `kyua report`, making the results easy to recover on the host.

The regression test suite covers a large number of FreeBSD features but is designed to complete quickly. So while it can help find bugs with relatively little effort, more intensive stress tests may be required. FreeBSD has several ways to further test kernel changes.

stress2

`stress2` is a large stress test suite maintained by Peter Holm. It contains many hundreds of regression tests for the core kernel's filesystem and memory management subsystems. The stress2 suite is included in the FreeBSD source tree, in `tools/test/stress2`, but is not part of an installation. To run the tests, assuming a source tree is available in the test system, run:

```
# cp -R ${SRCDIR}/tools/test/stress2 /tmp/stress2
# pw user add stress2
# cd /tmp/stress2
# echo stress2 | make test
```

The `stress2` suite requires at least several gigabytes of RAM and a large disk. It can take multiple days to complete but is an excellent way to test systemic changes to the kernel. The individual tests are located under the `misc` subdirectory and can be run directly.

syzkaller

Finally, `syzkaller` has emerged as an effective tool for exercising portions of the kernel reachable from the system call interface. Being a fuzzer, it is not particularly useful for proving the correctness of a change, but it is very good at triggering rarely executed error paths and so can help validate error handling code which may otherwise be difficult to trigger. It is also effective at provoking race conditions.

A detailed overview of `syzkaller` appeared in a previous *FreeBSD Journal* article. Setting up a `syzkaller` instance is a somewhat involved task. Documentation is available in the `syzkaller` repository for setting up a FreeBSD host to run `syzkaller` (which performs fuzzing using QEMU or bhyve VMs).

An alternative approach which automates many of the setup steps makes use of Bastille templates. Bastille is a system for deploying and managing jails on FreeBSD; Bastille templates allow one to run code and modify configuration in a running jail. A Bastille template for running `syzkaller` is available. To use it, first install Bastille and create a thin, VNET-based jail based on FreeBSD 13.0:

```
# pkg install bastille
# bastille bootstrap 13.0-RELEASE
# bastille bootstrap https://github.com/markjdb/bastille-syzkaller
# bastille create -V syzkaller 13.0-RELEASE 0.0.0.0 epair0b
```

This assumes that `epair0b` is Bastille's "uplink" interface; it can be bridged with a host interface to provide full network access.

Then, follow the setup steps documented in the bastille-syzkaller template's README to create a ZFS dataset for `syzkaller` and enable the required capabilities in the syzkaller jail. Finally, the template can be applied, and `syzkaller` started, with:

```
# bastille template syzkaller markjdb/bastille-syzkaller \
  --arg FREEBSD_HOST_SRC_PATH=${SRCDIR}
# bastille service syzkaller syz-manager onestart
```

Typically `${SRCDIR}` would be a git worktree referencing the branch to be built and tested; this worktree is `nullfs`-mounted into the jail. The kernel fuzzed by `syzkaller` can be rebuilt using the `build.sh` script installed by the template in the jail root user's home directory:

```
# bastille console syzkaller
# service syz-manager onestop
# sh /root/build.sh
# service syz-manager onestart
```

At this point, `syzkaller` starts a web server listening on port 8080, displaying crash reports and the progress of the fuzzing processes.

MARK JOHNSTON has been a FreeBSD user since 2010 and a committer since 2013. He currently works for the FreeBSD Foundation, where he spends time on improving the stability and performance of the kernel, and on providing code reviews.

PRACTICAL PORTS

Programmers Programming Potpourri

BY BENEDICT REUSCHLING

This column covers ports and packages for FreeBSD that are useful in some way, peculiar, or otherwise good to know about. Ports extend the base OS functionality and make sure you get something done or, simply, put a smile on your face. Come along for the ride, maybe you'll find something new.

One of Unix's strengths is that even though it involves a lot of typing, it grew a number of helper utilities over the years. People developed neat little shortcuts to avoid typing or repeating the same keystrokes. One such invention was the shell history. Why type something again when you can retrieve it from way back when you typed it successfully the last time. For the uninitiated, typing the right keystrokes to invoke searching in the shell history will make them fold a wizard hat in no time—mouth still open in astonishment. Once they learn how to do that, they might become less impressed. Then it is time to install misc/mcfly and race through your shell history just like Marty did in the movies. The intelligent search takes your current directory into account or the context within which you used the program and offers you the proper commands. It does not mess with your normal shell history file, allowing you to get comfortable with mcfly.

Why type something again when you can retrieve it?

Programmers like to listen to audio when hacking on the latest code. A soothing visualization of the sound waves hitting your ears has been popular since the days of Winamp. If you want the same on the console, audio/cava is there for you. No matter what you use: Pulseaudio, fifo (mpd), sndio, squeezelite or portaudio, jumping bar graphs will appear

to match your tunes. Sweet, but don't forget to wear headphones or your neighbors get angry. Talk about eavesdropping.

Programmers not only write code, but they also need to test it. Performance is still crucial for the user experience, even today with hardware the Apollo guidance computer would only have dreamed of. To ensure a code change did not make things run more slowly, benchmarks are used. One such benchmark for the commandline is benchmarks/hyperfine. It supports arbitrary shell commands for statistical analysis across multiple executions. A warmup phase ensures that caches don't interfere with measuring the right things. Outliers caused by some other program running in the background are detected and hyperfine can run with varying numbers of threads. The results are output in CSV, JSON, Markdown, and AsciiDoc. Your thesis basically writes itself these days...

Did you ever have the need to securely transfer a file or string like a password over to another computer? If SSH is too complex for you with its 740-character, public-key exchange, but you don't want to compromise on the security front, send your files through net/py-magic-wormhole instead. When sending a file, a short, humanly pronounceable character string is generated. Input this one-time key on the receiving side and the wormhole does its magic using Rendezvous Message Exchange and PAKE-based security. It even allows tab-completion for the key, saving you some keystrokes. The magic wormhole can also serve as a replacement for ssh-copy-id for the initial SSH key exchange. In the example below, I transferred my backup.zip over to another machine. On the receiving side, I entered the code and confirmed the file. Moments later, the file was on the other side, just like a 30,000 light-years trip through the delta quadrant.

```
$ wormhole send backup.zip
Sending 1.2 GB file named 'backup.zip'
Wormhole code is: 1-specialist-apple
On the other computer, please run:

wormhole receive 1-specialist-apple

Sending (<-[redacted]:60188)..
100%|[redacted]| 1.23G/1.23G [00:14<00:00, 87.6MB/s]
File sent.. waiting for confirmation
Confirmation received. Transfer complete.
$
```

```
$ wormhole receive
Enter receive wormhole code: 1-specialist-apple
 (note: you can use <Tab> to complete words)
Receiving file (1.2 GB) into: backup.zip
ok? (y/N): y
Receiving (->tcp:[redacted]:31976)..
100%|[redacted]| 1.23G/1.23G [00:14<00:00, 86.7MB/s]
Received file written to backup.zip
$
```

A common exercise for my undergraduate computer science students is to let them parse text files in Unix. Comma separated values (CSV) are still popular file types in these assignments. This usually happens before students get introduced to databases and SQL. If they knew before, they'd probably all use the next tool I'm introducing without ever looking back: textproc/csvq. Read, update, and delete CSV files to your heart's content. An interactive shell or commandline-mode did not let the developers stop there. Execution of multiple operations in sequence is possible in managed transactions. Variables, cursors and yes, temporary tables are also supported. Oh, did I mention that besides CSV, JSON is also supported in full UTF-8 and UTF-16 variants? Because why not?

Another tool is textproc/dasel (data selector), which supports JSON, YAML, TOML, and XML in addition to CSV. It can convert between the different formats, put new content in and delete existing entries. No need to learn a new tool each time a new language comes around. Chances are, someone has it already covered with "just another supported tool."

Developers usually have a lot of machines, both at work and home. They also want to have their familiar configuration everywhere, so that they are productive wherever they are. To stop copying the dotfiles (that hold all the configuration magic) from one machine to the next, there

is sysutils/chezmoi. It synchronized the dotfiles across machines in a secure way. A single command can pull down all these configuration files on a new machine. When changing settings on one machine, a single "chezmoi update" will ensure the change is present on another machine, too.

A quickstart guide and other documentation helps you use chezmoi in no time.

If you like syntax highlighting and line numbers in your output and wondered why cat(1) never evolved this feature, consider using textproc/bat. A cat clone with wings, capable of showing you non-printable characters, does automatic paging, and includes git integration.

Some developers like to brag about how much code they've written. But how much of it are comments and blank lines? If you want to find out, devel/tokei can tell you all about it. Tokei understands multi-line and nested comments and ignores comments within strings. It supports over 150 languages, colors, and is very fast, even with big projects.

Some developers like to brag about how much code they've written. But how much of it are comments and blank lines?

When I have not used a tool for a while, I tend to forget all the command line switches. Then I have to look up the man page and build my command line again (especially with multiple such flags). I should probably have written it down in a cheatsheet to save time. Something similar must have gone through the head of the misc/cheat developer. How about all there is to know about using tar? You can find this example on its github project page:

```
$ cheat tar

# To extract an uncompressed archive:
tar -xvf '/path/to/foo.tar'

# To extract a .gz archive:
tar -xzvf '/path/to/foo.tgz'

# To create a .gz archive:
tar -czvf '/path/to/foo.tgz' '/path/to/foo/'

# To extract a .bz2 archive:
tar -xjvf '/path/to/foo.tgz'

# To create a .bz2 archive:
tar -cjvf '/path/to/foo.tgz' '/path/to/foo/'
```

Very handy and sometimes I find some new nuggets of wisdom in there, too. Try it with commands like rsync. Run

```
cheat -l
```

to list all available cheatsheets. Of course, you can add cheatsheets of your own and share them with the community. People will still think you are a Unix wizard by sharing these, perhaps even more then.

And if you thought that bat was the only port named after an animal mentioned in this column, prepare to meet two new: dns/dog and dns/doggo. The latter was inspired by the former, but since the author did not know Rust (which dog is written in), he rewrote it in go (dog + go, you get the idea). Both are a modern CLI DNS clients (like dig) and support protocols like DNS over HTTPS (DoH), DNS over TLS (DoT), DNS over TCP/UDP, and DNSCrypt. JSON support is integrated in both dns/dog and dns/doggo and at least one of them supports multiple resolvers at once. I'll let you find out which, so that you can also enjoy the colors it produces. What a dark world we used to live in. But then LSCOLORS slowed things down too much, but that's a story for another time.

But while we're on the subject of colors, if you like them too much, why don't you mix some sysutils/lsd into your Unix work? This pretty ls alternative has both colors and icons and was written in Rust (is there anything left that isn't?). It certainly looks ... colorful. Don't blame me for headaches or other woes incurred by using any kind of lsd.

Back to the subject of this Journal issue, but staying with the colors: have you ever looked at your git diff output and wondered if that could be improved? Wonder no more, devel/git-delta comes to the rescue (even if you did not think you needed help). Layout and style for diffs is what delta allows you to spend hours on, just to get the right color scheme. It includes a pager and of course themes so that you don't have to start from scratch. Line numbers, side-by-side views, boxes with customized lines are all possible. Chances are, you will spend even more time reviewing code with this tool. See what they tricked you into?

When your disk is overflowing with code snippets, patches, and uncommitted work, it's high time to open sysutils/duff. No, not the fictional beer, the duplicate file finder. It styles itself to be a prettier du utility, but you'll be the judge of that. It certainly is fast and gives you an overview of where disk space was eaten the most. I'm sure that duffs grouping into local, network, and special devices gives sysadmins valuable information at a glance. Better run

```
git gc
```

once in a while to garbage collect some temporary files no longer needed.

Lastly, you could wake me up in the middle of the night and I would still be able to tell you the find commandline syntax to search for a file or directory. It might take me some time to spell out

```
find / -name "*foo*"
```

for you and I will be grumpy at you for waking me up. I value my beauty sleep, so I'll tell you about sysutils/fd. This tool is so easy to use that it only requires

```
fd foo
```

to list the results. Much easier, more sleep for me. Goodnight!

BENEDICT REUSCHLING is a documentation committer in the FreeBSD project and member of the documentation engineering team. He serves on the board of directors of the FreeBSD Foundation as vice president. In the past, he served on the FreeBSD core team for two terms. He administers a big data cluster at the University of Applied Sciences, Darmstadt, Germany. He's also teaching a course "Unix for Developers" for undergraduates. He is one of the hosts on the bsdnow.tv podcast.

Support FreeBSD®

Donate to the Foundation!

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

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

Your investment will help:

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

Making a donation is quick and easy.

freebsdfoundation.org/donate

FreeBSD™
FOUNDATION

2021 Events Calendar

BSD Events taking place through December 2021

BY ANNE DICKISON

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

OpenZFS Developer Summit 2021

November 8-9, 2021

VIRTUAL

The ninth annual OpenZFS Developer Summit will be held online, November 8-9 (Mon.-Tue.). The goal of the event is to foster cross-community discussions of OpenZFS work and to make progress on some of the projects that have been proposed. This 2-day event consists of a day of presentation and a 1-day hackathon.

November 2021 FreeBSD Vendor Summit

November 18-19, 2021

VIRTUAL

Join us for the online November 2021 FreeBSD Vendor Summit. The event will consist of virtual, half-day sessions, taking place November 18-19, 2021. It's free to attend, but we ask that you register with the eventbrite system to gain access to the meeting room and separate hallway track. In addition to vendor talks, we will also have discussion sessions.

FreeBSD Fridays

https://freebsdfoundation.org/freebsd-fridays/

Our next FreeBSD Fridays session will take place in December. Stay Tuned!
Past FreeBSD Fridays sessions are available at: https://freebsdfoundation.org/freebsd-fridays/

FreeBSD Office Hours

https://wiki.freebsd.org/OfficeHours

Join members of the FreeBSD community for FreeBSD Office Hours. From general Q&A to topic-based demos and tutorials, Office Hours is a great way to get answers to your FreeBSD-related questions.

Past episodes can be found at the FreeBSD YouTube Channel.
https://www.youtube.com/c/FreeBSDProject.