



ARM 64 is Tier 1

- **Datascience on FreeBSD/ARM 64**
- **FreeBSD on the Pinebook Pro**
- **ACPI Support for Embedded Controllers**

An ambitious company for ambitious people.

Juniper is changing what's possible in networking. We're going beyond building the networks customers expect—we're building the networks customers deserve. And the world is taking note. But to continue to excel, we have work to do. Change in our industry is accelerating. To power connections and empower change, we need radical thinkers, eternal optimists, and energized personalities. We need people like you.

The Junos Core Kernel team is looking for an ambitious Software Engineer with experience and passion for Kernel/Operating System technologies. Successful candidates will support and enhance the FreeBSD operating system. The team is responsible for designing networking/driver domains of our products.

<https://juni.pr/FreeBSDEngineering>



Editorial Board

- John Baldwin • FreeBSD Developer and Chair of FreeBSD Journal Editorial Board
- Tom Jones • FreeBSD Developer, Internet Engineer and Researcher at the University of Aberdeen
- Ed Maste • Senior Director of Technology, FreeBSD Foundation and Member of the FreeBSD Core Team
- Benedict Reuschling • Vice President of the FreeBSD Foundation Board and a FreeBSD Documentation Committer
- Mariusz Zaborski • FreeBSD Developer

Advisory Board

- Anne Dickison • Marketing Director, FreeBSD Foundation
- Justin Gibbs • Founder of the FreeBSD Foundation, President of the FreeBSD Foundation, and a Software Engineer at Facebook.
- Daichi Goto • Director at BSD Consulting Inc. (Tokyo).
- 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 • Treasurer of the FreeBSD Foundation Board, and lead author of *The Design and Implementation* book series.
- George Neville-Neil • Past President of the FreeBSD Foundation, member of the FreeBSD Core Team, and co-author of *The Design and Implementation of the FreeBSD Operating System*.
- Hiroki Sato • Director of the FreeBSD Foundation Board, Chair of Asia BSDCon, 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

- Publisher** • Walter Andrzejewski
walter@freebsdjournal.com
- Editor-at-Large** • James Maurer
jmaurer@freebsdjournal.com
- Design & Production** • Reuter & Associates
- Advertising Sales** • Walter Andrzejewski
walter@freebsdjournal.com
Call 888/290-9469

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@freebsd.foundation.org

Copyright © 2021 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

FreeBSD/arm64 is Now Tier 1

Arm's 64-bit architecture, AArch64, is now Tier 1 status in FreeBSD 13. Following from the 32-bit FreeBSD/arm we use the name "arm64."

Tier 1 means that the FreeBSD release engineering team will build and publish official releases for this architecture, in addition to the existing amd64 and i386. The security team supports the architecture with binary and source updates for vulnerabilities and errata updates. A full set of binary packages is available from the package team.

Arm publicly disclosed the details of the AArch64 architecture in October 2011, and Andrew Turner soon took an interest in a FreeBSD port. The FreeBSD Foundation began supporting the arm64 porting effort in 2014, with funding from Arm and Cavium. Collaborating with Andrew and Semihalf, Cavium's ThunderX processor was brought up as the first reference platform for FreeBSD/arm64. The Foundation contributed work on release engineering, tool chain support, and other areas.

Support continued improving over the years, but it took a confluence of factors to allow for the promotion to Tier 1.

Early on Hardware availability was limited, especially for server class machines. Today Ampere Computing eMAG and Altra CPUs, Arm's Neoverse N1 core, and AWS Graviton instances are well-supported. The Foundation purchased eMAG servers to build official package sets. Ampere Computing then donated additional servers. This allowed the project to support multiple simultaneous FreeBSD src and ports tree branches.

The tool chain was an early limiting factor. FreeBSD still used an older version of the GNU linker which did not support arm64, requiring awkward workarounds to use an out-of-tree linker. With Foundation sponsorship, the project migrated to using LLVM's LLD linker for all supported architectures in FreeBSD 13.

Thanks to tireless contributions from FreeBSD ports volunteers we were able to iterate on arm64 build failures; over 30,000 packages are now available.

In April 2021 on behalf of the core team I announced that FreeBSD/arm64 would be Tier 1 in FreeBSD 13.

I hope you enjoy the arm64 articles in this issue, and give FreeBSD/arm64 a try!

Ed Maste

Senior Director of Technology
FreeBSD Journal Editorial Board



5 **Datascience on FreeBSD/ARM 64**

By Maciej Czekaj

9 **FreeBSD on the Pinebook Pro**

By Jesper Schmitz Mouridsen

14 **ACPI Support for Embedded Controllers**

By Marcin Wojtas

3 **Foundation Letter**

FreeBSD/arm64 is Now Tier 1

By Ed Maste

21 **WIP/CFT: Lumina Desktop Calls for Developers**

By Tom Jones and JT Pennington

23 **Practical Ports**

How to Set Up an Apple Time Machine

By Benedict Reuschling

31 **We Get Letters**

ARM 64

by Michael W Lucas

34 **Events Calendar**

By Anne Dickison

Data Science on FreeBSD/ARM64

BY MACIEJ CZEKAJ

Recently, [ARM64 became a Tier I platform](#) for FreeBSD. Since Semihalf has a long history of supporting FreeBSD on anything ARM, it was a logical step to use it in production. The test bed was unusual, however, since it was not yet another Web server or NFS storage array (which we have plenty of already), but a full-fledged Data Science lab.

The task at hand was to run a large-scale simulation experiment on the [Marvell ThunderX2](#) ARM server. The simulation experiment resulted in the [scientific publication](#) and a chapter in the PhD thesis. The workload spans hundreds of CPU-hours for custom simulation software alongside the standard Open Source scientific toolkit, such as SciPy, Pandas, and Jupyter. The main bottleneck of the simulation system was RAM, while putting equal pressure on the disk I/O and data integrity. The software suite was originally developed for Linux and had to be ported to FreeBSD (by complying with POSIX).

ThunderX2 used in the experiment is a dual-socket 56-core ARM64 platform. The single CPU die has 28 cores in eight core complexes joined by the ring interconnect with shared L3 cache with cross-section bandwidth of more than 6TB/s. Each core may have up to 4 SMT threads totalling to 224 threads in the system. The 8-channel DDR4 interface for each die provides over 200GB/s of memory bandwidth for the whole system. The CPU dies are connected through CCPIv2 interconnect providing 600 Gb/s bandwidth. Looking at the specs, it seems to be the perfect target for memory-bound workloads.

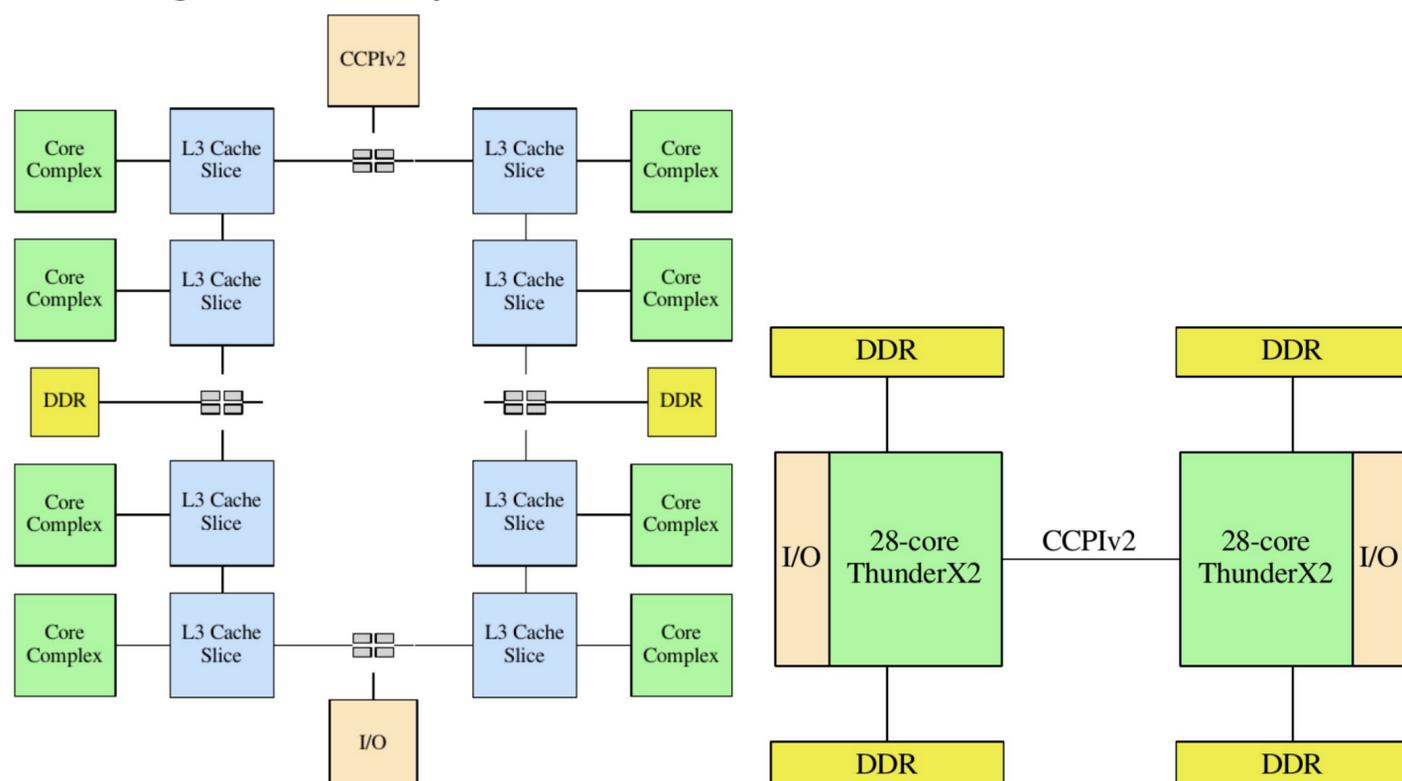


Fig1. The architecture of ThunderX2 system.

Originally used in the GNU Linux/x86 desktop environment, the simulation system had to be adapted to a parallel environment, possibly without too much programming effort. The central part of the system is a custom simulator software written in C++. The simulator accepts a recorded packet trace (PCAP stream) and produces a network flow database. The stream may come from a file or from another program (the mixer) which combines many packet streams together. The flow database is a custom binary format, which resembles the memory organization of a C table of structures. This format is both easy to serialize in C/C++ (by `fwrite()`) as well as easy to parse by Numpy (by `fromfile()`).

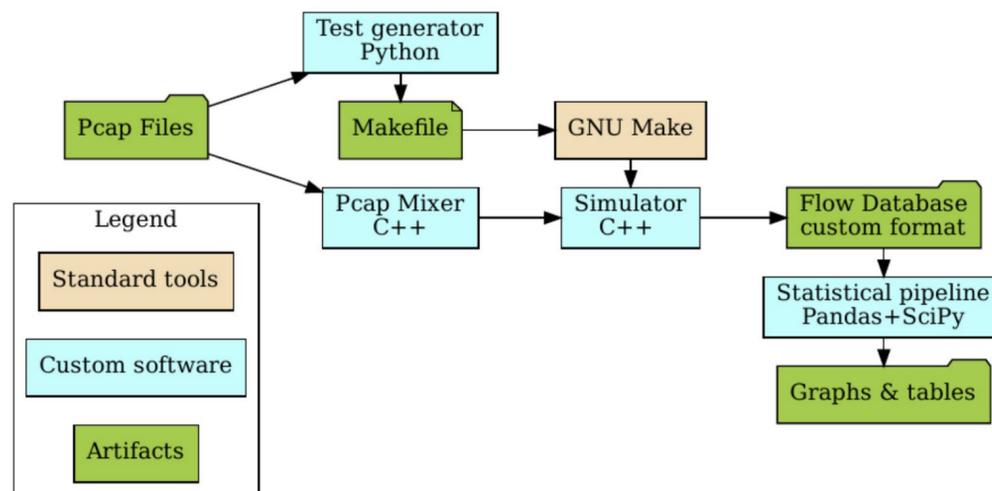


Fig2. The custom Data-Scientific pipeline executed on FreeBSD.

The next phase is controlled by statistical software in Jupyter Notebook. Each experiment produces millions of records occupying gigabytes of RAM, which mandates the use of an in-memory database for analytics in the form of Pandas DataFrame objects. The whole pipeline is described as a set of GNU Make job definitions.

The porting process from GNU/Linux to FreeBSD-12.2 was relatively simple. The C++ code base used mostly I/O system calls, which are part of the POSIX standard. Porting from GCC to Clang revealed a few issues about the code base itself, lending credibility to the common wisdom that using more than one compiler improves the code quality. The only functional issue was the usage of a hash function from the standard C++ library. The exact algorithm is implementation-dependent, so in order to keep the results reproducible, the hashing function source code must be provided. There were few performance issues with the C++ `iostream` library on FreeBSD. Granted, using text-based I/O was a design mistake in the first place, so the porting effort only amplified that inherent weakness. In summary, the porting of the C++ code proved to be the least concern and making it a multi-platform software improved the overall quality of the simulator.

The porting process from GNU/Linux to FreeBSD-12.2 was relatively simple.

Surprisingly, using the popular Python frameworks posed a bigger challenge than porting the C++ code. Popular scientific packages have many dependencies and usually are kept outside of a standard OS-specific Python stack. The essential challenge is to match the right version of Python, Numpy, SciPy, Pandas, Scikit-learn, and dozens of dependencies. In GNU/Linux the most popular way to resolve this conundrum is to use the binary Anaconda distribution. To my disappointment, the Anaconda dev team does not express any interest in supporting FreeBSD. The only alternative (apart from compiling everything from scratch) was to use Python Virtualenv. The fun started right away, when some of the packages were expecting GCC and

others assumed Linux-specific include paths. After the painful process, all the essential packages were compiled. This should not be a surprise that Python packages heavily rely on third-party C or C++ libraries. Many Python packages are only language bindings to libraries written in C. Each time the package is installed, the third-party dependencies must be recompiled. It is worth keeping in mind that the whole Python stack is not totally independent of the base system. Thus, upgrading the FreeBSD poses a risk of repeating the whole process.

If the deployment of the Python stack was so cumbersome, was it worth it? Ultimately — yes — due to parallel computing. By default, the computation on DataFrame objects is single-threaded. However, the [Pandarallel package](#) provides a seamless parallelization through multiprocessing. Though not perfect, as it mandates copying the data, the speedup is still significant for CPU-intensive computations.

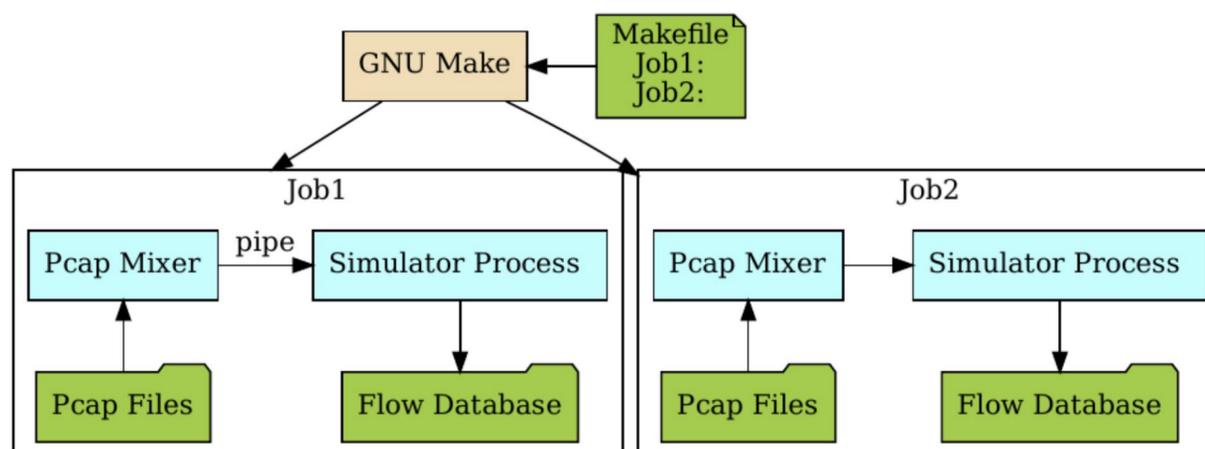


Fig 3. The parallelization scheme supervised by Make.

The simulation system was designed to be single-threaded. The packet processing job must maintain an order of packets, so the central algorithm must remain sequential. The only viable means to scale the workload to many CPU cores was to exploit the coarse-grained parallelism in the workflow itself. The workflow definition contains more than 500 independent jobs. Each job lasts from several minutes to an hour, with the memory consumption from 10 to 30 GB (just for the data structures, so that was essentially an un-swappable resident set).

Luckily, the Make utility is capable of supervising a fixed number parallel jobs though the ubiquitous “-j” option. The challenging part was to match the varying memory requirements of the jobs with the number of processes. To my knowledge, there are no build systems that try to limit the number of jobs based on the memory pressure. They all consider only CPU load. Thanks to the infamous FreeBSD OOM killer and the decades-old UNIX wisdom accumulated in the Make utility, that turned out to be easier than expected. Each time when the number of jobs exceeded the RAM capacity, the OOM killer would pick the most memory-hungry process. This resulted in wasted CPU time, but kept the whole system stable and responsive. Also, the artifacts produced by the killed process were removed by Make, so data integrity was not compromised. This behavior is contingent on the correct job definition, since only explicitly defined Make targets are deleted.

The final version of the system was running continuously for over a week with intermittent supervision from the operator. The high demand for disk I/O bandwidth was met by the use of SSD drives backed by the ZFS filesystem. The overall stability of the platform was proven beyond any doubts.

If the deployment of the Python stack was so cumbersome, was it worth it?

The final conclusions for role of the FreeBSD/ARM64 as a scientific platform can be drawn in few points:

1. The platform provides excellent stability. The low overhead of the system and minimalist distribution leaves plenty of room for CPU-intensive or memory-intensive tasks.
2. The I/O subsystem can keep up with the most demanding workloads as long as the backing storage is solid state.
3. Porting the software from x86_64 to ARM64 architecture is mostly as easy as recompiling, provided that the developer follows the best practices of creating portable code. In case of porting from Linux from the same architecture, Linuxulator provides Linux ABI for native binaries.
4. Complex software stacks which are not supported by the system package manager can pose some challenges, if there are no alternative package managers. It is best to use shortcuts, with pre-built environments based on jails. As usual with the Open Source community, the software needs a critical mass of users to draw the attention of developers.

This yet another FreeBSD success story is a testimony to the effort of many developers which solidified the ARM64 port to the point where using the system is as ubiquitous as any x86 machine.

MACIEJ CZEKAJ is a lead s/w engineer at Semihalf, specializing in high-speed networking applications and drivers. He is a contributor to the [DPDK project](#) where he claims the authorship of one of the first ARM64 Ethernet device drivers (VNIC on ThunderX ARM64 server). He received his Comp. Sci. PhD program at AGH University (Kraków, Poland) on high-speed network acceleration.



The FreeBSD Project is looking for

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

Find out more by

Checking out our website

freebsd.org/projects/newbies.html

Downloading the Software

freebsd.org/where.html

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

Already involved?

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

Help Create the Future. Join the FreeBSD Project!

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

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

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

The FreeBSD Community is proudly supported by



FreeBSD on the Pinebook Pro

BY JESPER SCHMITZ MOURIDSEN

The Pinebook Pro is a Rockchip rk3399-based arm64 laptop. It is not currently for sale (according to Pine64) due to global components shortage. You might already own one though, and have missed running FreeBSD on it. In this article, I will describe how to get FreeBSD running on it as a useful desktop. If you do not happen to have a PineBook Pro, the test image that I provide and the build steps also apply to the RockPRO64 board. (Except for U-Boot, do use the stock FreeBSD one for RockPRO64).

As I said, it is rk3399 based, but it is not RockPRO64 in a casing — it does have its own specific mainboard. So the official rockpro64 builds are not the way to go. There are some patches in review that do make the Pinebook Pro useful as a desktop, most notably, the drm-sub-tree of Emmanuel Vadot and the work of Ruslan Bukin who wrote about panfrost in an earlier issue. Also patches for working sound have been written by Alexander Tymoshenko. A lot of work has been done by the posters on the Pinebook Pro thread on forums.freebsd.org started by SleepWalker. It's worth a read if you want a closer look at the development history of getting FreeBSD to run on Pinebook Pro.

Supported Hardware

- The graphics stack with accelerated graphics by the panfrost driver work of Ruslan Bukin and related work by Emmanuel Vadot in the [drm-subtree](#).
- Sound recording and playback by Alexander Tymoshenko.
- emmc and sd-card are also fully supported.
- The SPI flash is detected, but is probably still hit by bug [244146](#).
- The PCI bridge is supported with an SSD, although ufs shows some weirdness in my tests.
- All CPUs are supported, but FreeBSD cannot make full use of big.LITTLE. i.e., the faster cores have to follow the frequency of the slower ones.
- USB-2 ports.
- Touchpad and keyboard — the touchpad is only working as a simple mouse though.
- Webcam works with webcamd.

Unsupported Hardware

- Wi-Fi and Bluetooth, and DP over USB-C.
- USB-c works as USB-c, if you turn it on with `gpioctl`. Do not try this if you are not totally sure on where to look in the device tree for the right pin setting and how to set it.

What Software Runs?

I have tested both sway and hikari / wayland and X11 with a couple of DEs. In the process, I found that openbox has an issue with the graphics stack. It does not really render anything correctly within the windows' frames. Luckily xfwm4 does not have any issues. Thus LXQt required a change from the default openbox to xfwm4. LibreOffice runs nicely. Electron stuff is amd64 only on FreeBSD, so you might miss some electron based applications e.g., vscode-oss. Sway needs to be reverted to version 14.1 otherwise you hit *"Cannot use DRM dumb buffers with non-primary DRM FD."* I suspect it is due to the setup where there are two card entries namely /dev/dri/card0 /dev/dri/card1 where only card1 has a render device, but I have not looked into it more closely. I choose instead to revert sway and wlroots to the last known good version on the Pinebook Pro.

Firefox on arm64 and perhaps other applications crashes often unless started with ASLR (address space layout randomization) disabled with `procontrol -m aslr -s disable firefox`. Also to test webgl in Firefox, you might have to set `webgl.force-enabled` to true in `about:config`. Since webcamd runs well and supports the built-in camera, I tested a `webrtc` call in Nextcloud talk with Firefox and it worked well. Screen sharing in the call worked as well, but only one window at a time. Vlc also does not like the `screen:///` capturing, so something with full-screen capturing is an issue at the moment. I also watched YouTube full-screen video without glitches. Note that since this work is based on 14-CURRENT, the default package repository does not build quarterly — so some packages might fail to build once in a while, and thus be missing.

The Booting Process

U-boot versions without video support i.e., without showing anything on the display right at boot time are too old to be useful. Also, I skipped a panel driver in the test image, so the panel must be turned on by u-boot. Backlight working also apparently relies on a recent u-boot. The one I have used most successfully is 2021.7 which is the one in the FreeBSD ports tree at the time of writing.

NetBSD noted that the panel breaks on warm reboot. They have a patch that I highly recommend since the panel seems to start in an awfully wrong — and probably very bad for it — state on a warm reboot. The NetBSD patch is adapted to the FreeBSD ports tree [here](#) and is in the test image.

If you start from emmc with a FreeBSD u-boot, you should know that it does not default to booting the SD-card. To boot from the SD card in that case, you should press any key to stop autoboot and type `run bootcmd_mmc1`. Note that the keyboard under u-boot is not too well supported. It does type, but you better type slowly. One can also disable the emmc totally and just test the test image from an sd-card directly. You then avoid u-boot version issues, but it is a little inconvenient to use the internal laptop emmc (fragile) kill switch. It is easy to open the laptop — the switch is just badly placed in my opinion. So if you have a stock manjaro or debian installation, you might have to upgrade in a way that also updates u-boot or to install FreeBSD to the emmc or use its kill switch as mentioned.

Also remember that the u-boot that boots should be the patched one.



There are some patches in review that do make the Pinebook Pro useful as a desktop

The Quick Start

I will later describe how to install from source, but you can easily fetch the test image from [github](#) along with modified packages described later.

If you have disabled your emmc or installed the patched u-boot for pinebook pro to it--you can find it [here](#). You are ready to go. Note that RELEASE(7) has per default created two unsafe users with password root, and the user freebsd with password freebsd. ssh is enabled as well. All defaults for arm boards but not so appropriate for the laptop setup. Also do not forget to add your own user to the video group, otherwise the graphics stack would not be permitted to access the graphics device nodes.

Getting Online

Network connectivity is, of course, a must, but built in Wi-Fi is unsupported as of now. I choose to use a wifi dongle with a chipset from man rtwn_usb. You might also install a wi-fi card to m.2 slot if you have the pci-bridge expansion.

I did not try the later myself though. Your cell phone in usb tethering mode can also bring your Pinebook Pro online. For a Wi-Fi dongle, you will have to know how to connect via command-line interface. I recommend you edit /etc/rc.conf right away. See the [handbook](#) for more information about that. Using your phone a dhclient ue0 should be enough.

How to Cross Build From Source on amd64

The test image is based on 14-current at commit c9e023541aef. To build it, you can use my PINEBOOKPRO.conf and release.sh at people.freebsd.org/~jrm/pbp

```
./release.sh -c arm64/PINEBOOKPRO.conf
```

But it takes a while to build. (About 1.5 to 2 hours on a GEN10 10 core Intel CPU.

Panfrost is still work in progress at the time of this writing, so I modified Ruslan Bukin's latest PR against the drm-subtree to not use continuous memory since the Pinebook Pro apparently runs low on free continuous memory under relatively heavy loads such as the webgl samples.org aquarium in firefox. I am building panfrost as a module since it crashes at boot if compiled as a device. To build the module you can chroot to the scratchdir of release.sh and in /usr/src with the following environment variables set

```
setenv TARGET arm64
setenv WORKSPACE /usr
setenv MAKEOBJDIRPREFIX $WORKSPACE/obj/
setenv ROOTFS $WORKSPACE/rootfs
setenv SRC /usr/src
setenv MAKESYSPATH $SRC/share/mk
```

use

```
make buildenv TARGET_ARCH=aarch64 BUILDENV_SHELL=/bin/sh
```

I did a Makefile and some compile time error fixing consisting only of adding prototypes and printf format string fixes. You change dir to /usr/src/sys/dev/drm/panfrost when in the buildenv. Then you can execute

```
make
make DESTDIR=$ROOTFS install
```

You can then easily experiment with continuous memory by changing

```
-      if (1 == 1)
+      if (1 == 0)
            panfrost_alloc_pages_iommu(bo);
    else
            panfrost_alloc_pages_contig(bo);
```

in `panfrost_gem.c`. It does not have a `sysctl` knob, it is a code change. See the discussion on `drm-subtree`, pull [#13](#).

Ports and Modified Packages

As stated before, I reverted `sway` `wlroots` and `hikari` to earlier versions. `libdrm` is modified in order to detect the `panfrost` with this [patch](#). `Hikari` also needs a small patch to fix its argument parsing [issue](#) `mesa-dri` and `mesa-libs` are also modified to compile the `panfrost` driver and enable `gles1` and `gles2`, i.e., to compile in the way `Ruslan Bukin` described in his article.

The packages are prebuilt for download as is a patch for the ports tree if you prefer to build from source. You can take advantage of the `pkg lock` feature to not get the modified packages reinstalled by `pkg` commands. Simply run `pkg lock <pkgname>`. On my system I have the following packages locked:

```
hikari-2.3.2
libdrm-2.4.109,1
mesa-dri-21.3.6
mesa-libs-21.3.6
sway-1.6.1_2
wlroots-0.14.1_2
```

Note for `RockPRO64` owners, do not forget to reinstall the `RockPRO64` `u-boot` from ports to the test image, and note that `sound` is not patched in.

JESPER SCHMITZ MOURIDSEN is a self-taught system administrator and developer currently employed as system administrator working with `OpenStack`. He is a `FreeBSD` ports committer, with `LXQt` as his main focus and co-author of `rtx(4)`. AFK he likes a ride on his bicycle and is fan of cycling.



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!



2022 Editorial Calendar

- Software and System Management (January-February)
- ARM64 is Tier 1 (March-April)
- Disaster Recovery (May-June)
- Science, Systems, and FreeBSD (July-August)
- Performance (September-October)
- Topic to be decided (November-December)

Find out more at: freebsd.foundation/journal

ACPI Support for Embedded Controllers

BY MARCIN WOJTAS

ARM64 is a single architecture that is used in an extremely wide range of products — it can be found in the smallest embedded devices, but also in mobile devices, enterprise units and even server-grade solutions. The support for the latter imposes certain standards, e.g., the way of booting, interactions with firmware or a platform description. It turns out this model also fits non-server devices. Let's see how they are supported in FreeBSD, with a focus on handling the embedded controllers in the ACPI world.

Dealing With the Problematic Legacy

When introduced almost a decade ago, the 64-bit variant of ARM directly inherited the ecosystem from its 32-bit predecessor, which had been reigning in the embedded market. However, the usual need of maintaining a fully customized board support package for each platform was a real burden for development of the new architecture. To some extent, the device tree (DT) adoption allowed for better portability and using a single kernel image for various devices, but it did not suffice to solve the problem entirely. This kind of description is very flexible, which was, unfortunately, often abused by vendors and resulted in inconsistent bindings over the time. Even today, it is not uncommon that the device tree blob for U-Boot differs from the one used for booting the OS (they describe the same hardware!), and often also lacks backward compatibility. With such constraints, reaching a long-term goal of a wide software ecosystem and multi-OS support would be problematic.

However, the solution was out there and existed for years. The interfaces used in the x86 world were adopted and extended for ARM64, namely the boot process, EFI, SMBIOS and ACPI. With the server-grade devices that comply with the standards and use proper firmware, it is now possible to install FreeBSD and other OSs or hypervisors out of the box, simply by using installer images. What about smaller, embedded platforms? Fortunately they can also leverage the rich ecosystem the same way. There are conditions though — the hardware must not deviate from the standards (at least not too much) and



When introduced almost a decade ago, the 64-bit variant of ARM directly inherited the ecosystem from its 32-bit predecessor.

there are also strict requirements related to firmware. The guidelines are gathered into specifications, consecutively: the [BSA](#) (ARM Base System Architecture) and the [BBR](#) (ARM Base Boot Requirements). Result — there are ARM64 platforms that can successfully boot the FreeBSD, Windows and multiple Linux distributions, using a single firmware image and ACPI description.

What is special about those devices? Compared to the servers, which traditionally have a significant amount of CPUs, DRAM and PCIE root complexes, in the embedded segment the SoCs also support a wide variety of controllers attached to their internal buses. Therefore, they are not discovered during PCIE enumeration, but require a different treatment. A hardware description must comprise an explicit reference to these interfaces, including the platform data that can be parsed and interpreted by the OS. Recently, the FreeBSD kernel's ability to obtain such information from ACPI tables was extended with some new features.

What is ACPI?

Before jumping to details, it may be worth briefly explaining what the ACPI is — it is an interface between the firmware and OS, used for describing and configuring the hardware. The [standard](#) has been developed for almost 3 decades and lists a number of [main concepts](#), i.e., various aspects of power management, thermal/battery handling, hardware configuration and embedded controllers' description. It also defines an [ACPI Source Language](#) (ASL), which among others allows for creating low-level hardware configuration routines. It is compiled to a bytecode — [ACPI Machine Language](#) (AML), that can be interpreted and executed by the kernel.

The information about a platform is gathered in so-called 'tables,' which are, in fact, a hierarchy of structures in the system's memory address space. The starting point of ACPI is Root System Description Pointer (RSDP) structure — it is configured by firmware and points to Extended System Description Table (XSDT), which further branches out to secondary tables. The first one is always Fixed ACPI Description Table (FADT) — it comprises various fixed-length entries that describe the fixed ACPI features of the hardware.

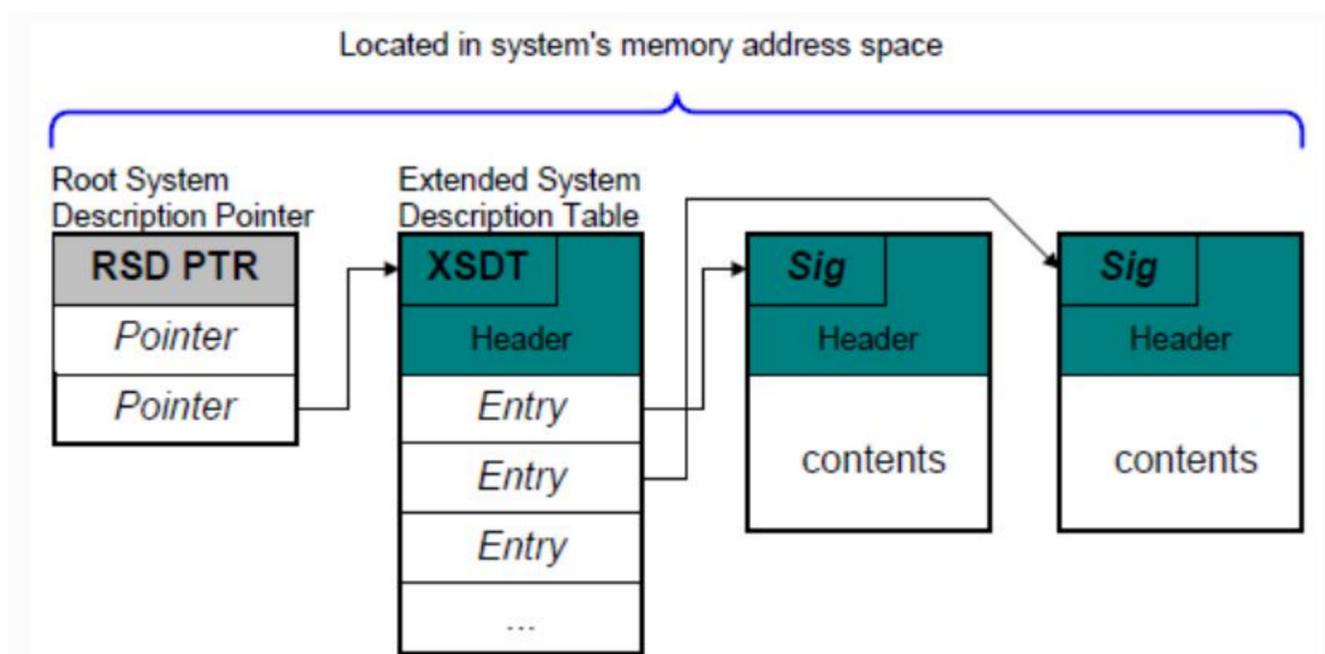


Fig1. Root System Description Pointer and Table.

Source: https://uefi.org/specs/ACPI/6.4/05_ACPI_Software_Programming_Model/ACPI_Software_Programming_Model.html#overview-of-the-system-description-table-architecture

The ACPI specification defines a number of [dedicated tables](#), however a couple of them can be considered as being more significant in the embedded devices context, e.g.,

- Generic Timer Description Table (GTDT)
- Multiple APIC Description Table (MADT)
- Processor Properties Topology Table (PPTT)
- Serial Port Console Redirection Table (SPCR)
- PCI Express Memory-mapped Configuration Space base address description table (MCFG)
- Differentiated System Description Table (DSDT)

The last of the mentioned tables is particularly important. The DSDT is always referenced by FADT and comprises the list of CPUs, power management features, PCIE root complex and all other embedded controllers description. It often comes with SSDT (Secondary System Description Table) — in single or multiple instances, this structure allows the programmer to logically split various functionalities in the platform description code.

The definitions of the above tables were extended to cover ARM64-specific values and types (e.g., interrupt controllers) — all gathered in ACPICA (ACPI Component Architecture). It is an open source reference code, used and supplemented by OSs. The FreeBSD is maintained to always be on par with the latest version of it. Let's check how the tables are handled in the ARM64 port.

ACPI for ARM64 — the Base Part

The ARM64 SoCs are described by the ACPI tables according to the standards, i.e., the timers and watchdogs are listed in GTDT, the interrupt controller can be found in MADT — currently only GICv2 and GICv3 are supported. Going further to the embedded controllers, the console is described by SPCR (and optionally by the additional DBG2 table) — using ARM SBSA UART (PL011) or the one compatible with 16550 is recommended, although in recent years more types from the ARM world have been added to [the list](#).

Description of the PCIE controller is more complex and must be enclosed in the MCFG and DSDT/SSDT tables. For ARM64 the only allowed type is the one fully compatible with the standardized ECAM generic, supported by [pci_host_generic_acpi_driver](#). It is recommended that the new designs comprise an unmodified version of it in the silicon, but for existing products, it is often not possible. Because of that, handling a deviation from the standards is now allowed in the mentioned FreeBSD driver, using the [configuration space access quirks](#). Another solution would be to support a mechanism of executing low-level routines from the firmware via the Secure Monitor Call Calling Convention (SMCCC) interface — currently it is available for [Raspberry Pi 4](#), but this option remains unimplemented in FreeBSD.

Handling of the Embedded Controllers

Embedded controllers that are connected to the SoCs internal bus can be handled twofold ways in the ACPI tables. One option is using 'methods' (instructions) compiled to AML, so the OS can interpret and execute them directly, which is the case of e.g., [thermal management](#), [SMBUS](#) or [GPIO](#). Other devices or subsystems that are not explicitly defined in the ACPI specification need to be described by the standard objects that are available for parsing by the OS and obtaining all necessary hardware resources required by the kernel drivers. The latter solu-

It is recommended that the new designs comprise an unmodified, generic version of PCIE controllers in the silicon.

tion is a key to support non-server ARM64 SoCs in ACPI and is already present in the FreeBSD kernel.

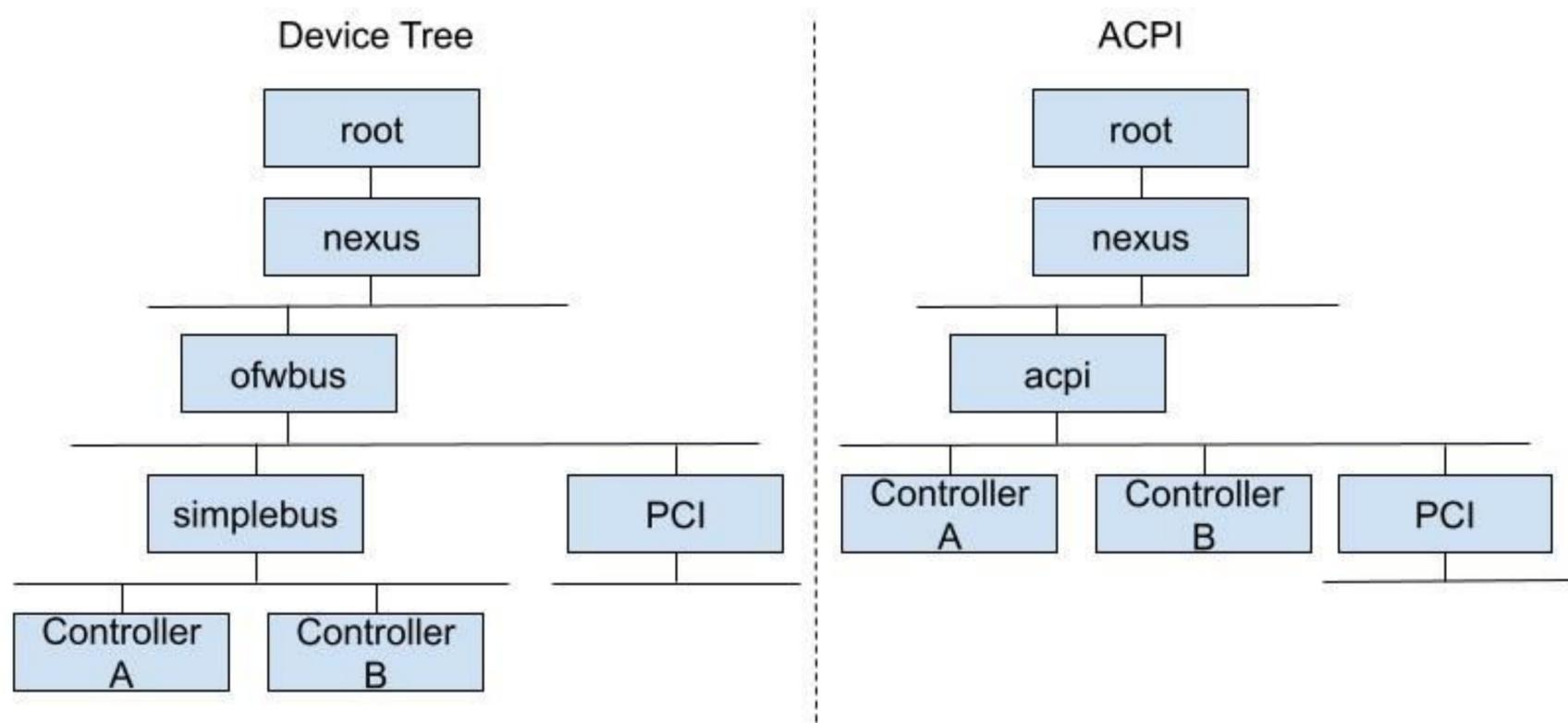


Fig 2. High level comparison of example FreeBSD bus hierarchies in ACPI and Device Tree worlds.

In high level, the FreeBSD bus hierarchies of embedded controllers are similar for both ACPI and DT worlds (ref. Fig. 2). It is helpful for designing the device drivers, as the platform data structures can be filled likewise in each case during the kernel initialization phase. The probed drivers can be later matched by the ACPI `_HID` field value, which can be treated as an equivalent to the compatible string known from the Device Tree. The other standard types of resources are also handled in an analogous way.

The first two types of ARM64 embedded controllers supported by ACPI in FreeBSD are [USB](#) and [SATA](#). The latter is interesting, because it is matched with a driver in a bit of a different way, i.e., by a device class value (ACPI `_CLS` object; ref. Listing 1).

```

Device (AHC0)
{
    Name (_HID, "LNR0001E") // _HID: Hardware ID
    Name (_UID, 0x00) // _UID: Unique ID
    Name (_CCA, 0x01) // _CCA: Cache Coherency Attribute
    Method (_STA) // _STA: Device status
    {
        Return (0xF)
    }
    Name (_CLS, Package (0x03) // _CLS: Class Code
    {
        0x01,
        0x06,
        0x01
    })
}
  
```

```

Name (_CRS, ResourceTemplate () // _CRS: Current Resource Settings
{
    Memory32Fixed (ReadWrite,
        0xF2540000,          // Address Base (MMIO)
        0x00030000,          // Address Length
    )
    Interrupt (ResourceConsumer, Level, ActiveHigh, Exclusive, ,, )
    {
        CP_GIC_SPI_CPO_SATA_H0
    }
})
}

```

Listing 1. Example AHCI controller description in [ACPI table](#)

The FreeBSD XHCI and AHCI drivers expect fully generic descriptions in DSDT/SSDT. An example of the former is presented in Listing 1. It contains objects referring to a unique ID, information about cache coherency and memory/interrupt resources. All deviations, such as a non-standard register configuration, clocks or power management handling have to be implemented and pre-configured by firmware.

Customizing the ACPI Description

What if the controller requires a custom binding handled by its own, dedicated driver? Until recently it was possible in FreeBSD only in the DT world, using the nodes' properties. However, the ACPI specification defines an optional object called `_DSD` ([Device Specific Data](#)), that can contain the same information. Leveraging the FreeBSD bus hierarchy (ref. Fig 2.), a new generic solution was [designed and implemented](#), to support obtaining controller specific data in a description-agnostic way. Additional helper functions were introduced:

- [device_get_property](#)
- [device_has_property](#)

They allow access to device specific data provided by the parent bus in a way that the consumer driver can execute exactly the same code path, regardless of the system booting with ACPI or DT. This solution was [later extended](#) to cover various types of properties available in both cases.

An example of the above was implemented in the SD/MMC subsystem, both in a [generic code](#) and a driver for Marvell Xenon controller. The latter was divided into three files: [common](#) part and small pieces responsible for attaching either via [ACPI](#) or as a child of [simplebus](#). Apart from different `DRIVER_MODULE/DEFINE_CLASS_1` macro usage, the latter comprises additional parsing of the regulators and card detect GPIO pins, whereas in ACPI these are set up by firmware.

The FreeBSD XHCI and AHCI drivers expect fully generic descriptions in DSDT/SSDT.

```

&ap_sdhci0 {
    compatible = "marvell,armada-cp110-sdhci";
    reg = <0x780000 0x300>;
    interrupts = <27 IRQ_TYPE_LEVEL_HIGH>;
    clock-names = "core", "axi";
    clocks = <&CP11X_LABEL(clk) 1 4>, <&CP11X_LABEL(clk) 1 18>;
    dma-coherent;
    bus-width = <8>;
    /*
     * Not stable in HS modes - phy needs "more calibration", so add
     * the "slow-mode" and disable SDR104, SDR50 and DDR50 modes.
     */
    marvell,xenon-phy-slow-mode;
    no-1-8-v;
    no-sd;
    no-sdio;
    non-removable;
    status = "okay";
    vqmmc-supply = <&v_vddo_h>;
};

```

Listing 2. Marvell Xenon SD/MMC controller in Device Tree

```

Device (MMCO)
{
    Name (_HID, "MRVL0002") // _HID: Hardware ID
    Name (_UID, 0x00) // _UID: Unique ID
    Name (_CCA, 0x01) // _CCA: Cache Coherency Attribute
    Method (_STA) // _STA: Device status
    {
        Return (0xF)
    }
    Name (_CRS, ResourceTemplate () // _CRS: Current Resource Settings
    {
        Memory32Fixed (ReadWrite,
            0xF06E0000, // Address Base (MMIO)
            0x00000300, // Address Length
        )
        Interrupt (ResourceConsumer, Level, ActiveHigh, Exclusive, ,, )
        {
            48
        }
    })
    Name (_DSD, Package () {
        ToUUID("daffd814-6eba-4d8c-8a91-bc9bbf4aa301"),
        Package () {

```

```

Package () { "clock-frequency", 400000000 },
Package () { "bus-width", 8 },
Package () { "marvell,xenon-phy-slow-mode", 0x1 },
Package () { "no-1-8-v", 0x1 },
Package () { "no-sd", 0x1 },
Package () { "no-sdio", 0x1 },
Package () { "non-removable", 0x1 },
    }
}
}

```

Listing 3. Marvell Xenon SD/MMC controller in ACPI

Listings 2. and 3. show example DT and ACPI nodes of the same controller instance, in order to demonstrate the similarities of both descriptions. Thanks to the new FreeBSD kernel methods, the controller can successfully operate in all firmware configurations.

Conclusion

With the recent additions to the FreeBSD kernel, the contemporary ARM64 SoCs used in the embedded products can be supported with a similar set of features both with ACPI and the Device Tree. The hierarchical representation of custom controllers turned out to be flexible enough for most kinds of devices and subsystems also in the ACPI case. There are examples that confirm it is possible even with more sophisticated network controllers and the generic MDIO layer. Now there are no limits for FreeBSD to follow this path, especially that the bus architecture allows doing it in a clean and elegant way, as demonstrated in the SD/MMC example.

MARCIN WOJTAS is Head of Engineering at Semihalf and also a FreeBSD src committer (mw@). He is passionate about embedded software and hardware, and contributor to a number of open source projects, including Linux kernel, Tianocore EDK2 and TF-A.



WIP/CFT: Lumina Desktop Calls for Developers

BY TOM JONES AND JT PENNINGTON

The free desktop space has a no shortage of options when it comes to desktop environments. They range from small minimal environments with a window manager and some utilities for decoration all the way up to competitors for Mac OS or Windows in the form of Mate, KDE and Gnome.

In the free desktop space there is a shortage of projects that are freely licensed and developed with BSD as an explicit target. The Lumina desktop fills this space. Lumina was originally started by Ken Moore to be a BSD-licensed desktop environment for TrueOS. It can be run on any OS and with the right support it targets a FreeBSD-based platform.

This allows Lumina to support FreeBSD specific features and to couple well to FreeBSD interfaces for the desktop environment.

Lumina development is now headed by JT Pennington who you might know as the magic behind the excellent BSDNow podcast (as a host, I am allowed to sing our praises). JT was involved in the PC-BSD and TrueOS projects and it was there he met Ken Moore. JT picked up the project when Ken's work commitments made it hard for him to contribute as much time as he wanted.

JT, speaking for Lumina has asked for help to carry the project forward. He has ideas for projects which are a little beyond the time he can contribute — these are big steps forward for the desktop environment.

Open Project Requests

In a blog post on the Lumina Website [<https://lumina-desktop.org/post/2022-02-08/>], JT wrote requests for help with three areas:

- Port build system from Qmake to Cmake
- Lumina File Manager rewrite
- Lumina 2.0 Window Manager

Port Build System from Qmake to Cmake

Qmake is QTs build system for projects, but it is starting to act as a barrier for porting to Linux distributions (and is probably a headache in FreeBSD, too). Help here would increase the user and testing base for Lumina and a larger testing base means a better desktop environment.

Lumina File Manager Rewrite

The Lumina File Manager is unique because it is able to integrate with the OS quickly. It offers zfs-specific features giving you access to snapshots in a simple and convenient view, something no other file manager offers.

Lumina-FM is ready for a rewrite, it is time to learn from the good features it has now and to add more customization. JT's blog post details some ideas he has for a better file manager, these

Lumina was originally started by Ken Moore to be a BSD-licensed desktop environment for TrueOS.

WIP/CFT: Lumina Desktop Calls for Developers

include showing break downs of disk usage for file hierarchies and better thumbnail caching that understands network drives.

Lumina 2.0 Window Manager

The 2.0 version of the Lumina Window Manager doesn't need grand new features, instead it needs to be updated and enhanced to the state where it can replace the current Fluxbox Window Manager which is the basis of Lumina.

First the Lumina Window Manager needs to be developed to feature parity with Fluxbox, then the next step is to add in some features obviously missing from Fluxbox such as the ability to resize a window from any corner and modern features like snapping to screen corners.

Advanced features like Wayland compatibility can come later, but first the Lumina project needs a usable window manager.

How to Contribute

Lumina uses QT as its basis for its windowing toolkit. To be able to help you will need to know (or want to learn), C++ and the QT interfaces. The improvements to the file manager and the build system will also require knowledge or a willingness to learn a lot of intricate details about zfs and cmake respectively.

Both Lumina and its web site are on github [github.com/lumina-desktop/lumina], the project welcomes contributions there in the form of pull requests for bug fixes and new features and issues to report bugs or to make feature requests.

Lumina is an important part of the BSD eco system, if you have time to contribute in any form, then the Lumina project will be happy to hear from you.

The Lumina File Manager is unique because it is able to integrate with the OS quickly.

TOM JONES wants FreeBSD-based projects to get the attention they deserve. He lives in the North East of Scotland and offers FreeBSD consulting.

How to Set Up an Apple Time Machine

BY BENEDICT REUSCHLING

Apple's time machine has been around for a number of years to back up Macs over the network. It is a simple solution to set up and runs in the background without bothering the user too much. Users can retrieve their files from previous versions of their file system or even restore their whole computer to a new one to continue where they left off.

Apple initially offered separate hardware called time capsules for this task, but in recent years focused more on backing up to their paid iCloud solution.

Some users may not want to trust their file system contents to some other computer out there beyond their reach and control. Luckily, setting up a FreeBSD system to act as a time capsule on the local network is still supported. In this article, we'll walk you through such a solution.

A time capsule is basically a service listening on incoming connections from the time machine protocol and then stores the data submitted (the latest backup delta) on the local file system. Fans of OpenZFS trust that the built-in data integrity features keep their data intact, which is why we're combining time machine with OpenZFS here. As we are trusting our most valuable files from the Mac to our FreeBSD system, we want to ensure that we can retrieve it again one fateful day when we desperately need it back.

Another consideration is noise and power use. Since this local time capsule system is supposed to be running 24/7 for backing up and retrieving the files, we don't want a noisy solution that also draws a lot of power. Surely, one could limit the service to only run during the hours when we are actually awake and use our Macs. Some people keep a separate time machine at work for redundancy reasons, so that could be limited to the typical 9 to 5 work hours. Nevertheless, we don't want to increase our energy bill too much. By the same token, running a time machine at home or in the office under our desks with noisy fans will disturb colleagues and family members alike. The solution to both problems is to use an ARM embedded board for the task. Not only are they cheap (cost-wise), but also come in a small form fac-

Luckily, setting up a FreeBSD system to act as a time capsule on the local network is still supported

tor which does not take up much space like a big server would. Pretty much all of them come without fans and are practically noiseless when running. Since ARM focused their chip development on energy efficiency, you'd be surprised how few Watts are needed to juice these boards. Finally, you don't need much computing horsepower to run a time machine server, as it mostly does I/O. I should also mention the cost factor: buying a small ARM board plus some external storage should still be cheaper than buying a time capsule from Apple. Maybe you'd like to donate some of the money saved this way to a BSD Foundation of your choice to support the continued development of the operating system?

I have run a time machine backup on a Raspberry Pi 3 with external storage connected to it for a while without any issues. You can build this solution on any recent ARM board that FreeBSD supports (i.e., boots and can install packages), it does not necessarily have to be a Raspberry Pi. The required configuration is not too complicated and once it is running, you can forget about it as it does not need constant attention. As long as you have some external storage, you can start making backups to it from your Mac. The reason why you want to use an external storage is that the flash on the boards is typically limited in capacity and may wear out when constantly written to. You can start with a single external disk and later create a ZFS mirror out of it when the next paycheck arrives. When the disk space gets low in your FreeBSD time capsule, the time machine protocol automatically removes older backups to make space. No intervention is necessary, it all runs on its own.

In this article, we assume that you have the board running with FreeBSD, connected to the network and external storage. It should be powerful enough to run ZFS on it, as this is my preferred solution. It will run just fine with UFS, so use that if your board's hardware is not strong enough for ZFS. Create a ZFS pool as outlined in the FreeBSD handbook to get started. We'll later create the necessary datasets on it as part of the setup.

First, we need to install two primary packages with dependencies that provide the time machine service:

```
# pkg install netatalk3 avahi-app
```

Avahi allows the discovery of the time machine service on the local network in an easy way. The Apple file server protocol, Apple Talk in version three, is what time machine is built on. Together, they will make the configured time capsule available on the network so that Macs can find and automatically back up their data to it. The installation should not take too long, depending on how powerful your ARM server is (or any kind of server you run this on). You can also decide to run the service in a jail, dedicating a jailed dataset to it for the backups. I'll leave that as an exercise for you to keep this article simple enough.

The backup files from the Mac should be stored for each individual user in their own directory. This way, we keep them separate from each other and allow other people to make backups as well. Let's assume our ZFS pool is called backup (what's in a name?). The following command creates a new dataset to hold all users' backups in their own directories:

```
# zfs create backup/timemachine
```

We also set some zfs options in case they are not set on the pool level already.

```
# zfs set atime=off backup/timemachine
# zfs set refquota=1T backup/timemachine
# zfs set reservations=1T backup/timemachine
# zfs set compression=zstd backup/timemachine
```

The first option disables the file system access time, which we don't need to run time machine successfully. The `refquota` and `reservation` options ensure that only 1 TB of pool storage will be allocated for the backups, but that they are guaranteed to be available no matter how much space non-timemachine files on the pool take up. Adjust this to fit your pool size and needs. Don't set this too low, though, or you won't be able to backup much from your Macs and older backups get deleted more often. The final option activates compression on the dataset. Be mindful of the compression algorithm set in the last option. Your ARM board may not be able to provide that much CPU power for the compression, so change this to a different algorithm or disable compression altogether. On my time machine, the compression ratio is low (1.01x currently), but it may depend on the type of files you back up from the Mac and how well they can be compressed.

Next, let's create a group called `timemachinists` that are allowed to use the time machine service. You don't want your noisy neighbor using your time machine for his backups, but maybe allow a new colleague in your office to back up her Mac, too.

```
# pw grouadd timemachinists
# pw groumod timemachinists -m bcr
```

Check the result of this operation using `pw groupshow timemachinists`. I'm the only user in that group at the moment. You can also pick a different group name as long as you reference it from the config file we'll show below. Each user should have their own dataset, so let's create one for myself and set proper permissions:

```
# zfs create backup/timemachine/bcr
# chown bcr:timemachinists /backup/timemachine/bcr
# chmod 0700 /backup/timemachine/bcr
# chmod 0777 /backup/timemachine
```

I allow only myself access to the `bcr` dataset, only the other `timemachinists` group members should be allowed to peak into my precious backups. Although the files are stored in a database format and not as they are on my Mac, I'm not taking any chances. On the `/backup/timemachine` dataset, the permissions are wider open for the service to properly access it. Now let's see how we reference this group and the mount point from the time machine configuration file. It is located in `/usr/local/etc/afp.conf` and contains the main time machine settings. To get started, the following configuration changes should be made:

```
[Global]
afp listen = 10.20.30.40
uam list = uams_dhx.so,uams_dhx2.so
mimic model = TimeCapsule6,116
disconnect time = 1
unix charset = UTF8
cnid scheme = dbd
file perm = 0640
directory perm = 0750
hostname = "Time Machine"
hosts allow = "10.20.30.0/24"
zeroconf = yes
log file = /var/log/afp.log
```

```
log level = default:info
vol preset = TimeMachine
vol dbpath = /var/netatalk/CNID/$u/$v/
```

```
[TimeMachine]
path = /backup/timemachine/$u
time machine = yes
valid users = @timemachinists
```

Here is what the options do, line by line, starting in the Global section. The “afp listen” line defines the host name or IP address of the machine that the service runs on and listens for incoming connections. This is the address that users will configure in the time machine settings on their Macs later. We chose 10.20.30.40 as an example here, adjust it to fit your own local network.

The “uam list” refers to the user access methods allowed. The ones we use here are the default ones that allow Diffie-Hellman eXchange protocol (versions 1 and 2) encoded passwords. Other possible options allow guest access (undesirable for most people) and Kerberos V, which may be interesting in a corporate setting.

If you care about what icon is displayed on your connecting Macs, the “mimic model” option is for you. If you’re feeling nostalgic, you can display a PowerBook (which predated time machine by some years) here. However, the Time Capsule option makes the most sense to use here.

A more useful option is the “disconnect time”. It may happen that connections get interrupted, and the time machine service keeps the session open, preventing further connection attempts with a “volume in use” error message. This option cleans up disconnected sessions after 1 hour. Adjust this if you get this error often, but you should not encounter it much when using the setting used here.

Specifying the server character set to the default UTF8 is done by the “unix charset” option. This should only be changed when you’re certain that you need it, otherwise leave this option alone.

A “cnid scheme” is what is being used for the backend of the volume. This is the database that keeps track of what files have been backed up, if and when they changed, and other administrative information. You don’t have to know much about this to run the time machine service, keeping the default dbd is fine for most people.

Both the “file perm” and “directory perm” define with what kind of permissions the files and directories from the connecting clients are stored, respectively. This could be more restrictive than the original permissions or less, but both cause more headaches than you have painkillers for, so leave the ones we suggested here.

The “hostname” parameter is the description of your time machine that the Mac users will see when clicking on the time machine icon in their status bar when correctly configured. Pick-

Although the files are stored in a database format and not as they are on my Mac, I’m not taking any chances.

ing a funny description here results in status messages between backups like “Last backup on black hole.” You’re easily amused, aren’t you?

Remember your noisy neighbor that should not be allowed to use your precious time machine storage for his Mac? The “hosts allow” option limits the service to certain hosts or networks, restricting everyone else. As much as your colleagues in the surrounding offices may complain, only the people sharing the same four walls around you can back up if you set it properly.

We want to use “zeroconf” to ease our burden to manually help computers find the service, so we set this option to yes.

Logging activity of the time capsule is a good idea to debug problems users may encounter when trying to use the service. The “log file” specifies the location of the log file and the “log level” the detail and number of messages being logged. The ones we use here are fine for day-to-day operations, while also not wearing out your board’s storage with too many writes. When debugging an issue, temporarily set it to warn or error for more details and restart the service.

With “vol preset” we define a section in the same configuration file for settings concerning the volume. Multiple volumes can be configured this way in the same configuration file, but for our purposes, a single one is enough.

The last option in the General section is “vol dbpath”. Remember our different users we may have tapping this service in the future? With this setting, each user gets their own independent settings in a subdirectory. My own volume may be called “bcrvol” and the settings for it get stored under `/var/netatalk/CNID/bcr/bcrvol/` (replacing the variables for my username and volume, respectively). The benefit of separating these by user and volume and not having one directory for all users is that if things get corrupted, this is limited to only one user. This will not happen often, but better safe than sorry (especially when it comes to backups).

We’re still not done yet with this file, but the last options are straightforward enough. The section we reference here from the [Global] section deals with who is able to access and where they should be allowed.

A volume “path” is where a backup from a certain user is stored. Since we don’t know who this will be down the road and adjust this file each time a new user comes along, we use the `$u` variable here. This way, my own files end up in `/backup/timemachine/bcr` and correspond to the datasets we created earlier. Don’t get confused here: the path is the file system path where the backed up files from the Mac will later reside. The “vol dbpath” option that used a similar variable is where the administrative database with meta information about the backup is stored. Even better, you don’t have to visit both directories at all and can forget about them once the service runs.

The “valid users” line specifies users or, in our case, the timemachinists group (distinguished from users by a leading @) that are allowed access. See how flexible this is? In the future, when we want to allow Susan Sunshine to also backup her Mac, we just need to create a user for

Logging activity of the time capsule is a good idea to debug problems users may encounter when trying to use the service.

her, add it into the timemachinists group, and create a dataset under `/backup/timemachine` with proper permissions for her.

```
# pw groupmod timemachinists -m susan
# zfs create backup/timemachine/susan
# chown susan:timemachinists /backup/timemachine/susan
# chmod 0700 /backup/timemachine/susan
```

No need to revisit this configuration file again, because variables and the group definitions will pick up the new user automatically. In case you still need to, each configuration line is described in more detail in `afp.conf(5)`. Let's save and close this file.

Just as Darth Vader felt a tremor in the Force in the presence of his old master, we want to automatically notify the Mac clients of the presence of your time machine service using Avahi. To that end, we create a new file in

```
/usr/local/etc/avahi/services/afp.service
```

and add the following contents:

```
<?xml version="1.0" standalone='no' ?><!--*-nxml-*-->
<!DOCTYPE service-group SYSTEM "avahi-service.dtd">
<service-group>
  <name replace-wildcards="yes">%h</name>
  <service>
    <type>_afpovertcp._tcp</type>
    <port>548</port>
  </service>
</service-group>
```

This basically defines where the AFP protocol will listen on (the "afp listen" line from `afp.conf` plus the port 548 defined here). With this file in place, we only need to enable and start the services to finish the server side of the setup.

```
# service dbus enable
# service avahi_daemon enable
# service netatalk enable
```

In case you're wondering, dbus came in as a dependency during the package installation. It needs to run alongside the other services: avahi for network discovery and netatalk since time machine works only with the Apple Filing Protocol (AFP).

Let's start these services right away to move on to the client side.

```
# service dbus start
# service avahi_daemon start
# service netatalk start
```

Check the output of

```
# sockstat -l
```

for the daemons listening on their respective ports and look into `/var/log/afp.log` for any errors.

On the Mac that should be backed up by the newly created time machine, we need to make sure that Time Machine recognizes this (non-Apple official) network volume. To do that, run the following in Terminal.app:

```
$ sudo defaults write com.apple.systempreferences TMShowUnsupportedNet-
workVolumes 1
```

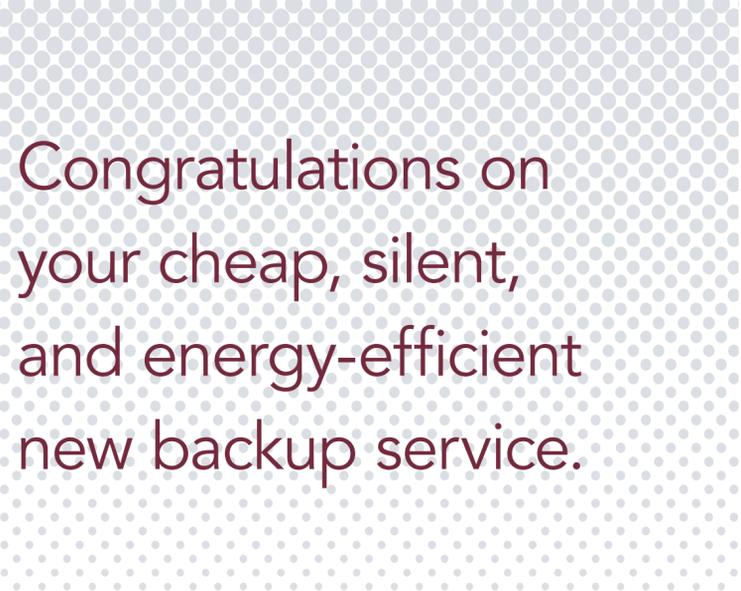
Next, go to the Finder, hit CMD-K (or select “Go to” from the menu and then “Connect to server...”) and enter the following:

```
afp://10.20.30.40
```

Substitute my example with the address or hostname that you configured in `afp.conf` and hit the connect button. Enter your username and password that you have on the time machine server (bcr in my case). If all goes well, the share will be mounted over the network and will appear in your left finder sidebar as an empty drive. When the mapping does not work, check the server’s log file again and make sure you have the proper IP address and user credentials.

Open the system preferences for time machine and click on “Add or remove backup volume”. In there, you should see your mounted share from the server. Select it and for extra protection, check the “encrypt backup” option. This is the only time where you can do this, not afterwards! Yes, you could also use ZFS encryption for the dataset, but I’m fine with this setting for my backup needs.

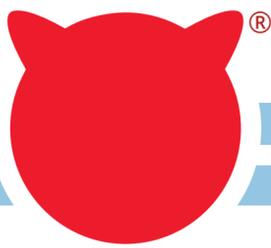
Time machine will now start creating the initial backup, which will take a long time depending on the number of files on your Mac and their size. Read through the other articles in this Journal to pass the time. Once the initial backup is finished, the service will disconnect automatically and reconnect in regular intervals to copy files that have changed. Test your backup by creating a small text file, wait for it to be backed up, then delete it. Act like you just accidentally deleted an important file and restore it using the Time Machine dialog, letting out a dramatic sigh of relief. That’s all, congratulations on your cheap, silent, and energy-efficient new backup service.



Congratulations on
your cheap, silent,
and energy-efficient
new backup service.

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. Benedict is one of the hosts of the weekly 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.
freebsd.foundation.org/donate





**Dear FreeBSD Journal Letters Column Answerer,
There's a bunch of excitement over how ARM64 is now Tier 1. How should I use it? Should I immediately switch my desktop, my servers, and my media server over to ARM64? Just from reading this very issue, I've gotten really excited about it. How fast should I move?**

—Searching Out Amazing Machines

"My day is my own," I thought when I woke up this morning. "I can lounge around and think about the glorious success that will inevitably descend upon me if only I can keep the bit about the wombats, the school bus, and the algae bollard under wraps, which should be simplicity incarnate because nobody knows what a bollard is except for a handful of literati who know nothing of wombats. I should probably write down a sentence or two, something about how the information reported by hard drives is not merely deceitful but actively treacherous, just so I can claim that I'm doing real work instead of wandering about the house listening to wire recording mix tapes from the 1930s and wondering how I can keep the squirrels from nesting in my emergency pants. It's not that I need pants all that often. They make bad days wholly terrible, like when I need to leave the house to find a gelato service that understands the difference between promising and boasting. The current one isn't it. Maybe the next."

And then your letter arrives, SOAM, ruining an otherwise perfect day.

On the plus side, I get to crush your hope. That's always nice.

All operating systems have an idea of tier 1 architectures. This means that the operating system can be installed on that architecture, that it runs, and that updates will be available to fix the inevitable bugs. Crash dumps will receive the same mixed attention as those of any other major platform. That sounds fine, right?

The problem is that sysadmins don't run hardware. We don't even run operating systems. We run applications. FreeBSD might be tier 1 on ARM64, but that doesn't mean your application is. Sure, there's lots of packages available. Many ports build. Perhaps even most. But just because the code compiles doesn't mean that it works let alone interoperates with whatever malware you're passing off as an application stack. People are using ARM64 for real work out in the real world, but that doesn't mean that you can. You thought Linux-isms were bad? Wait until you get a look at Intel-isms. Sure, people are working on their applications to make them work on ARM64, but the change in architecture has opened vast new realms of bugs. The obvious bugs have been found. What remains are the highly specific ones. Your environment is highly specific. Logically, these bugs all belong to you.

Many technologists claim that ARM64 is inevitable. The only inevitabilities are core dumps and that orange—and—green rash on my neck. People who should know better tout the advantages of ARM64 as if anything in computing could ever be improved when we all know that the pain never goes away, only changes. Install an ARM64 web server, and you'll discover tiny changes in behavior will put your application at risk. The people pushing ARM64 keep babbling about "reduced power consumption" and "open platforms," and they're extremely stubborn, so I suspect that they'll eventually get their way. A change of pains is as good as a rest.

So, what do you do?

You could start by not writing letters to this journal. That would have been an improvement.

Failing that, you should prepare for failure.

Your vital application runs on ARM64? Great... for some value of "great."

You can't start using it yet. Even if you set up an ARM64 system purely for testing and turn on all the debugging you can find so you can catch application errors and submit bug reports, you almost certainly have no idea what normal looks like. Your idea of normal is a quiet help-desk phone. When your brand-new ARM64 system starts spewing cryptic messages about locks and updates and whatever sort of flimflam the developers yammered that made your organization decide that this particular group of lies would solve their problems, you'll have no idea if these are normal or not.

You're starting in the wrong place.

Application developers rarely design useful logs. A few intend to. Many design logs that they find useful, which is not the same as useful to you. You need to know what normal logs look like, so you can recognize abnormal ones.

Start playing with ARM64 by going to your legacy environment, full of AMD64 or MIPS or even (ugh) i386 hardware. Make a list of your vital applications. For each one, figure out how to gather debugging data. Wholesome systems send everything to syslog, where you could distribute it into individual log files as needed, but many modern developers have abandoned this healthy

practice in favor of randomly selected logging systems that happen to conform to their prejudices so you'll have to (ugh, ugh) read the documentation. Some sysadmins have centralized logging servers where they can perform analysis of messages from every system they manage, but they are overachievers, and we will discuss them no further. Worst case, find a convenient log4j instance on the public Internet and dump all your debugging there. They won't mind.

While you are digging up logging configuration information on your every critical application, make a list of how to file bug reports on each and every one. It's much easier to do this before a notably vexing bug raises your blood pressure and triggers your brain's wired-in "kill one developer or massacre them all?" decision-making circuits.

Now that you have a baseline for comparison, you can install your ARM64 system and see what happens. Don't get me wrong, it's going to fail. As always, the question is how it will fail. Your ARM64 systems will have logs stuffed with cryptic meaningless messages. Fortunately, you already have functioning servers that have their own cryptic, meaningless log messages. You can compare the two and, with luck and perhaps a simple conjuration at an abandoned crossroads during the new moon, sort out messages that indicate your actual error.

**Application
developers rarely
design useful logs.
A few intend to.**

Prepare a bug report.

Send it to the application developers.

If they answer, they will almost certainly wonder why you're using their application in a way that was never intended, but that's exactly what UNIX is for, so ignore the sniveling. Work the problems, one after the other, until your application truly runs on ARM64. That's how open source software works. It's people like you, doing the grunt work of polishing and problem-solving, so that future decades of lazy bastards like myself can reap the rewards.

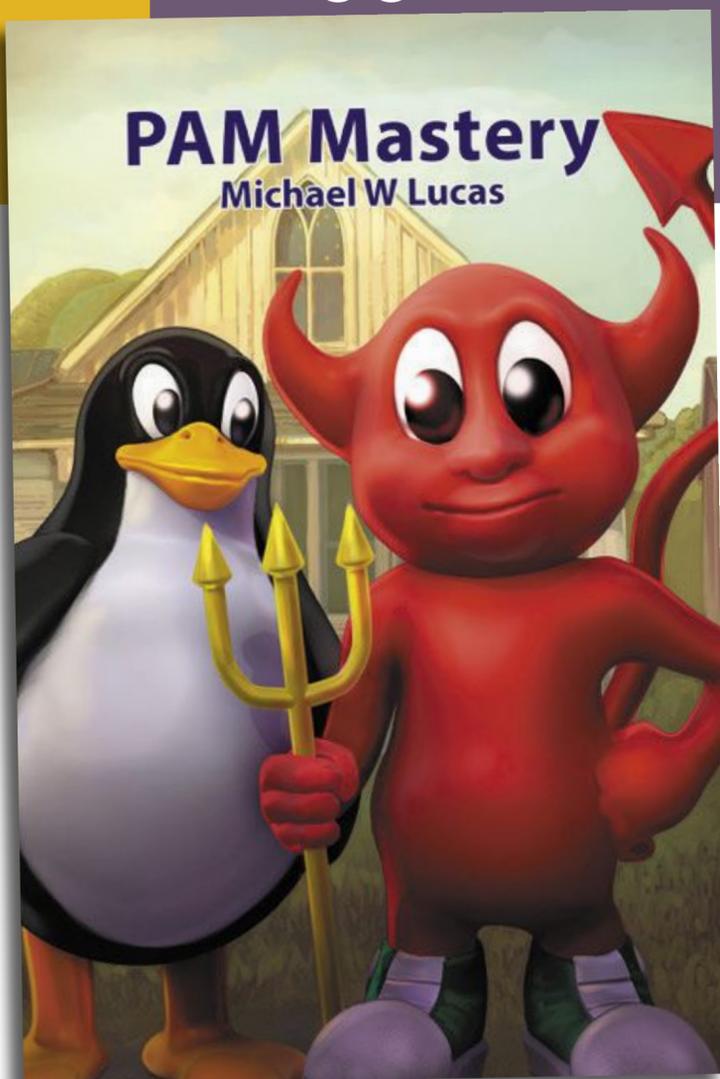
If that's not enough of an answer for you, too bad. I hear a squirrel gagging in the garage, so I know where my emergency pants are. I should probably wash them one year.

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



MICHAEL W LUCAS has spent too many decades debugging hardware platform migrations. His latest books include *DNSSEC Mastery* and *\$git sync murder*. He's also released *Letters to Ed (1)*, a collection of the first three years of this very column. Why he thinks you'll pay good money for something you get for free right here, we have no idea.

Pluggable Authentication Modules: Threat or Menace?



PAM is one of the most misunderstood parts of systems administration. Many sysadmins live with authentication problems rather than risk making them worse. PAM's very nature makes it unlike any other Unix access control system.

If you have PAM misery or PAM mysteries, you need PAM Mastery!

"Once again Michael W Lucas nailed it." — nixCraft

***PAM Mastery* by Michael W Lucas**

<https://mwl.io>



Events Calendar

BSD Events taking place through July 2022

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.



BSDCan 2022

June 1-4, 2022

VIRTUAL

<https://www.bsdcan.org/2022/>

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



2022 FreeBSD Developer Summit

June 16-17, 2022

VIRTUAL

<https://wiki.freebsd.org/DevSummit/202206>

Join us for the online June 2022 FreeBSD Developer Summit. The event will consist of virtual, half day sessions, taking place June 16-17, 2022. It's free to attend, but we ask that you register with the conference system to gain access to the meeting room. In addition to developer discussion sessions, we will also have vendor talks.



SCaLE 19x

July 28-31, 2022

Los Angeles, CA

<https://www.socallinuxexpo.org/scale/19x>

SCaLE is the largest community-run open-source and free software conference in North America. It is held annually in the greater Los Angeles area. Roller Angel will also be hosting a FreeBSD workshop during the conference.

FreeBSD Fridays

<https://freebsd.foundation.org/freebsd-fridays/>

Stay tuned for new episodes.

Past FreeBSD Fridays sessions are available at: <https://freebsd.foundation.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>