LINUX FOUNDATION

TRAINING

The History of Embedded Linux & Best Practices for Getting Started

Henry Kingman

July 2013

A Linux Foundation Training Publication
www.training.linuxfoundation.org

DISCLAIMER
The Linux Foundation frequently works with subject matter experts on Linux educational pieces. In this publication, famed Linux Devices' editor Henry Kingman has created a fun history and background on embedded Linux; we hope you enjoy Henry's unique perspective on this fast-moving industry. The opinions found in this publication are Henry Kingman's alone and don't necessarily reflect those at the Linux Foundation.

Overview

This paper offers historical perspective and advice for those considering or planning to embed Linux. Before we plunge in, let's define some terms:

- By "Linux," we mean any stack that includes a Linux kernel. Besides the kernel itself, stacks typically include a great deal of other software – typically the GNU C libraries, some graphics stack (like xorg), and the vendor's particular software "value add." However, a Linux device may include little more than a bootloader, bootstrap loader, and initramfs (i.e., a Linux kernel and a module or two running on a ramdisk).
- By "embedded device," we mean any device with a microprocessor, exclusive of PCs and servers. This describes the vast majority of computerized systems, since by most analyst estimates, PCs and servers consume only 1-5 percent of all microprocessor chips. The rest go into that vast, diverse, hard-to-generalize-about category known broadly as the "embedded" or "device" market. Many such systems, perhaps two thirds, have 8- and 16-bit "microcontrollers" that do not support "rich" 32-bit OSes like Linux. Even so, Linux-capable devices with 32-bit processors still far outnumber PCs and servers. Just consider all those devices everywhere around us: mobile phones, tablets, set-top boxes, networking gear, surveillance cameras, auto infotainment systems, HVAC and factory control systems, remote sensing stations, cellular radio towers... and on and on and on.

Across the vast diversity of systems encompassed by this definition, Linux is a strong contender -- if not the de facto standard. By the mid-Otts, most analysts were recognizing Linux as the top embedded OS in terms of design starts and design wins – if not devices shipped outright.

Okay, It's Big - But is It For Me?

Before diving into some of the historical forces that led Linux to world device domination, let's consider some of Linux's limitations. That way, those whose embedded OS needs may lie elsewhere can rule out Linux without further reading.

- As noted, Linux requires a 32-bit processor or greater. Linux's rise to world device domination coincided with the advent and widespread proliferation of low-cost, highly integrated 32-bit "system-on-chip" processors, many using RISC architectures such as ARM, MIPs, PowerPC, and SuperHitachi, to name just a few.
- In order to use Linux's popular "memory protection" and "virtual memory" capabilities, the processor hardware must include a memory management unit (MMU). However, a version of Linux dubbed "microcontroller Linux" (or more commonly "uClinux", pronounced "you-see-Linux") can run on MMU-less 32-bit and up processors, albeit obviously without the benefit of Linux's famous memory protection features (more about that later).
- Unlike some traditional embedded RTOSes (real-time operating systems), Linux requires a filesystem -- if only the aforementioned initramfs option. Linux can mount filesystems residing on storage or in RAM. To accommodate their filesystem(s), most Linux devices have at least 2MB each of memory and storage. For systems with memory management hardware (aka, "MMUful systems"), those figures grow to 8MB/16MB respectively. Most embedded Linux systems have considerably more.

- At the other end of the spectrum, Linux's default ext4 filesystem has incredible theoretical size limits (that may in practice be limited by associated tools like ext4defrag). Furthermore, more mature alternative filesystems like xfs and jfs offer something for the most storage-hungry enterprise/datacom devices.
- Linux has supported processors with 64-bit memory addresses since around 1995, when Linus Torvalds completed the DEC Alpha port. Alpha was the first non-x86 port completed "in-tree" (a PowerPC tree also saw feverish development at the time). These early porting efforts laid the foundations for Linux's subsequent explosion in devices, which tend to use non-x86 architectures due to their inevitably lower electrical power requirements.
- On processors with 64-bit memory addressing, Linux supports a nearly unimaginable RAM size. On 32-bit processors, the kernel is usually built nowadays with support for physical address extension (PAE). PAE supports physical memory sizes up to 64GB, rather than the 4GB theoretical max that 32-bit processors would otherwise top out at.
- Linux may not be certifiable for the very highest security levels (EAL 5 and greater). This is because the kernel itself is large, even in its most stripped-down form. That, in turn, makes truly exhaustive code line audits simply impractical. This limitation prevents adoption in esoteric markets, like top secret military devices, or medical devices deployed inside the human body. That said, Linux has an excellent reputation for security and reliability, and sees widespread use in military applications, such as drone fighters and other autonomous vehicles. It is also widely used in medical gear that collects and processes data from hardware sensors implanted inside patients. Linux has even seen space duty, in several British and U.S. (NASA) rocket ship launches!
- A standalone kernel.org Linux kernel (without any real-time add-ons) may not be practical for the most stringent hard real-time requirements, like if you need interrupt latency guarantees lower than 2-10ms (depending on your hardware). The threshold is always trending downward, however, as kernel.org continues to assimilate more and more code from out-of-tree real-time projects. Furthermore, where true hard real-time really is needed, a multitude of free and commercial projects exist to "pick up the slack" (so to speak).

Is Linux For You? Look To the Past

If you are considering Linux for your project or product, it may behoove you to have at least a passing understanding of the historical forces that led to Linux's rise in devices. That way, you stand a better chance of "riding the horse in the direction it's going," to paraphrase singer-songwriter Chuck Pyle (aka "The Zen Cowboy"). Or "skating to where the puck is going," to paraphrase the great Wayne Gretzky.

So, just why has Linux gotten so popular in devices?

Computerized devices began to appear in the 1960s, for example with shoe phones used by Maxwell Smart and other CONTROL agents.

Heh, kidding. It was actually the 70s when digital devices proliferated. Digital watches, microwave ovens, VCRs... these primitive systems used 8- and/or 16-bit microcontrollers. They lacked anything that, by modern standards, resembled an operating system, instead relying on "state machines," "executives," or other types of software stacks custom-written according to the requirements at hand.

Where a device did have an actual OS, it was invariably written in-house, usually from scratch. Amazingly, many such "roll your own", "home-grown", and "in-house" RTOSes (real-time operating systems) remain in service today, in industries from telecom to HVAC to factory control and automation. Hey, if it works, why replace it? Especially considering the guargantuan investment of resources required to develop, test, deploy, and fine-tune your very own pet operating system.

The 1980s saw the arrival of the first “general purpose” commercial operating systems designed especially for embedded systems. Sometimes, these are called “off-the-shelf” RTOSes. Hunter and Ready came out with their Versatile Real-Time Executive (VRTX) in 1980, pioneering a new market segment, according to most computing historians.

Today, VRTX is owned by Mentor Graphics. It supports ARM, MIPS, PowerPC, and other RISC architectures, with or without memory management hardware. VRTX still has some prominent “design wins” -- the Hubble space telescope, as well as the software-controlled radio layer in Motorola mobile phones.

A raft of other general purpose RTOSes followed. Several achieved greater commercial success than VRTX. By the mid-1980s, Wind River’s VxWorks had emerged as the market leader. Others found their specialized niches, like Green Hills Integrity for super high-security applications, or QNX Neutrino, for running your special real-time program alongside more or less normal Unix apps.

What Was So Great About Off-the-Shelf RTOSes?

The primary advantage, compared to writing your own, was that by using an off-the-shelf RTOS, you could get to market faster. It just saved you all this time. All you had to write were the applications, your “value add.” The OS and drivers were already up and running – maybe even smoothly! – on your development board.

Okay, you might pay extra for “smoothly.” But still.

As it turns out, in the device market, time-to-market is the single most important thing. Consumers of all stripes are fickle. Device buying trends are short-lived. Today’s high-margin plum piles up on tomorrow’s commodity discount table. Late-comers to the party: please turn out the lights.

Consumer trends are longer lived in stodgy industrial embedded markets, where products may have 20-year or longer life cycles. Yet, the first companies to reach such markets still gain the upper hand.

How important is time-to-market? Midway through my second week writing LinuxDevices.com, I vowed never to use the phrase “time-to-market” in a single story there, ever. If I did, even once, I would have to include it in every single story from then on. It’s really that important. In the embedded market, time-to-market matters so much that it literally goes without saying.

Commercial RTOSes lowered time-to-market, and as a result, they became huge. Market researchers began to value the commercial RTOS industry, including embedded development tools, at over $1B annually, starting around the year 1999.

Okay, you might be thinking: in 2012, Apple clears that much every single week, mainly through the sale of devices. But back then, a “billion dollar market” was considered quite a significant milestone.

Coincidentally, 1999 was the first year you could buy a commercial, off-the-shelf embedded Linux distribution aimed at devices and embedded applications.

The First Embedded Linux Vendors

Opinions vary on which company actually reached the embedded Linux market first. I’ve been given to understand, by people who ought to know, that Caldera may have been first to achieve significant economic returns in the embedded market. However, MontaVista was the first to formalize a product offering for the space, and as a result is considered by most as the first embedded Linux vendor.

MontaVista shipped its Red Hat-based “Hard Hat Linux” in 1999. Later that year, on July 20, Caldera spun out Lineo, which marketed its SuSE-based embedded Linux offering as “Embedix.”

MontaVista, incidentally, was co-founded by Jim Ready. The same Jim Ready who co-founded by Hunter and Ready Systems, creators of the first off-the-shelf RTOS. So right from the start, embedded Linux had some pretty smart money behind it.

Today, Lineo survives in Japan, and serves the consumer electronics industry. MontaVista, too, remains alive, albeit as a wholly owned but reportedly independent subsidiary of chip and network appliance specialist Cavium Networks.

Cavium claims to have been founded by some of the talent liberated when DEC disbanded its Alpha chip team. Cavium also employs substantial engineering resources in India. Most of its chips use multi-way (typically 4- to 16-way) MIPS cores, together with a lot of networking-specific co-processors. Most of the microprocessors the company produces seem to go into its own line of low-cost, extremely power-efficient networking gear, most of which targets the small to mid-size enterprise market.

During the early Otts, a raft of other companies began to jump on the embedded Linux bandwagon. By 2001, there were half a dozen embedded Linux tool and OS startups. By 2003, Linux was even being adopted almost universally by traditional RTOS vendors. Sure, Linux itself was free, but commercial vendors could offer value-adds like tools integration, support guarantees, license indemnity, and so forth.

The largest RTOS vendor, Wind River, began dabbling in the Linux development tools market as early as 2003. When it added Linux versions of its flagship RTOS stack products around 2004, no one could deny any longer that Linux truly had “arrived” as a viable commercially supported embedded RTOS option.

Many Eyeballs

The first embedded Linux vendors based their products on Linux distributions like Red Hat and SuSE that were already in widespread use. The testing and integration these distributions got in the server and workstation markets made for an easier-to-use, more stable product, the thinking went.

This strategy aimed to leverage “Linus’s Law,” the theory proposed and named by Eric S. Raymond. It states that “given enough eyeballs, all bugs are shallow.” In other words, the more use software gets, the more likely that bugs will be found and fixed. The fewer bugs software has, the more people are likely to use it. This virtuous cycle is sometimes likened to the “network effect,” whereby the value of something like a telephone network increases as more and more people join it.

At the time, though, applying widely used, general purpose software in devices clashed wildly with accepted embedded practices. Many device engineers wrote Linux off, seeing it as too large, too unpredictable (it lacked hard, real-time determinism), and too hard to test using automated testing routines. Devices must have toaster-like reliability, the thinking went, and the last point, in particular, raised concerns.

Linux had two things going for it, though.

Linux is MMU-tiful

Traditional embedded RTOSes of the 1990s primarily used a “flat” memory model. The engineer would allocate a chunk of memory, and their application was expected to stay inbounds, with little or no supervision from the OS or from the underlying hardware. When code failed to behave as expected, it might over-right some memory registers belonging to another application, or even the RTOS. Thus, a crashing application in such a system could easily cause the whole system to lock up.

Such bugs were notoriously difficult to find. If they emerged after a product shipped, an expensive recall could result. Thus, most devices of the day underwent exhaustive automated testing that exercised each routine throughout its complete range of possible variable values.

Well-tested embedded systems with flat memory models were often elegant, minimalistic, efficient, and very stable. However, the cost was high, due to the involvement of specialized personnel (QA engineers) and lengthy testing processes that posed an inconvenient barrier to market entry.

Linux, meanwhile, was a "virtual memory OS" that, thanks to its desktop origins, was ready to leverage hardware memory management units (MMUs). At that time, the early Otts, MMUs were increasingly common, even on very low-cost 32-bit embedded processors.

As a virtual memory OS, Linux and the applications that run under it do not address actual physical memory addresses. Instead, each process receives a separate virtual memory address space, with MMU hardware enforcing the separations. This makes it impossible for one application to write to or erase memory not previously allocated to it. The practical result is that a virtual memory operating system – sometimes called an operating system with "memory protection" – is much better insulated from application crashes.

The main advantage to using a VM OS like Linux in an embedded system is that the burden of testing may be reduced considerably. Today, the embedded Linux systems that power digital cameras, printers, or television sets may routinely endure application faults, without the user ever knowing. Linux keeps running, and simply reboots the program guide or menu or player application, with the user noticing little more than a slight pause.

It is easy to see the appeal of virtual memory, compared to the rigorous testing approaches of earlier device OSes. In the early days of embedded Linux, Linux's "memory management model" was its most-talked-about technical feature.

The downside is that virtual memory OSes require considerably more storage and memory. That could be a deal-breaker in cost-sensitive, high volume markets. So, pretty early on, a version of Linux for MMU-less processors appeared. Called uClinux ("You See Linux" or rarely "Micro-controller Linux"), its development was driven largely by an Australian company selling low-cost enterprise networking gear. Today, uClinux remains widely on "deeply embedded" systems.

Best Practice: Unless you have significant time, experience, and staff to rigorously test your application code for high reliability and security in all foreseeable use cases, consider employing a processor with an MMU, and a version of Linux (i.e., not uClinux) that supports memory management.

The other big, often discussed Linux feature, early on, was the possibility it gave companies to achieve better vertical integration of their product development process. What does that mean? The best way to explain it may be to look at the relative weakness of traditional RTOSes in this regard.

The RTOS Achilles Heel(s)

First, there is this company, in your product, extracting royalty. The more widgets you sell, the more you pay the RTOS supplier.

Compared to services, products typically offer the potential for higher margins, and thus greater "scalability." With royalties retarding the rate of return, though, scalability is lost, and the potential reward for each product development investment becomes proportionally smaller.

Linux, meanwhile, is royalty-free. Even if you buy your Linux from a commercial vendor, you typically pay **by the number of developer seats, not the volume of products shipped**. That's a crucial distinction, and one that allowed products with Linux inside to scale upwards in profitability on more business-friendly curves.

As noted, another way to describe this is that using Linux allows companies to increase their vertical integration, the total amount of the product stack that they control.

Note: A couple of Linux vendors have experimented with royalty-bearing price models, through the years, particularly in high-end markets, like high-availability telecommunications. However, the author is not aware of this approach ever being particularly successful.

The second Achilles heel that product companies see when they look at an RTOS vendor is: **there is this company, in your product, and you have to rely on them for everything**. Drivers, tools, library extensions... "All your base are belong to us," as geeks of the late twentieth century liked to say, for reasons forgotten in the mists of time. Something about a poor translation of a pirated game.

Unless it signs all the NDAs, and pays another big chunk for a source code license, a product company may find itself wholly dependent on an RTOS provider, for just about everything. Please take a number... so much for time-to-market.

Meanwhile, with Linux, you have the source, at no extra charge. The world's best documentation can't compare with being able to read, debug, and tweak the code yourself.

Best Practice: Try to take ownership of as much of your firmware stack as possible. Such vertical integration makes you less prone to costs and delays introduced by additional supplier partnerships.

And that brings us to the last big puzzle piece. We've looked at Linux's technical edge as a VM OS, and how it let product companies gain better vertical integration. There was one other little thing that, though discussed last, was definitely not least.

The Social Contract

Royalty-free, portable, stable, flexible, MMU-tiful... Linux changed the game, where device OSes are concerned. Right?

Well no – we actually had all that already. By the time Linux really started to hit its stride in devices, NetBSD was already highly portable, mature, and completely open source.

The difference was: with Linux, you had to give back. The GPL said, essentially: All One or None, Exceptions Universally, Absolutely None.

Okay, actually, that was Dr. Bronner. But it's pretty close to the same thing.

And who knows? GPL author Richard M. Stallman must have spotted a few Dr. Bronner's soap labels in the MIT shower rooms of the day. Could it be that Tux owes his tails to that peppermint-loving German emigrant, seventh-generation soap maker, and insane asylum escapee, who was obsessed with spreading the message of universal cooperation through soap labels?

Linux is licensed under the GPL, version two. That means that if you write a new, say, device driver, you're obliged to share it. Get better real-time? Hand it over. Add a protocol stack? Great, we'll take it.

Use Linux in a shipping product? You just activated the GPL's "distribution" obligations... fork over the source that you built everything from, with your customizations, and by the way, don't forget your build scripts. It's right there in the license. (In practice, the embedded Linux device community could stand to do a better job of including build scripts, in my opinion).

So what was the upshot? Pretty much solely because of this license, almost instantly, the number of drivers available for Linux was greater than for the "more free" open source RTOSes, like NetBSD. Almost instantly, real-time performance was better. Practically overnight, there were more sample implementations and reference applications, and it just got easier and faster to build your device on

OS. More drivers meant more consumer choice that lowered your parts bill-of-materials, too.

And then as Linux's popularity grew, even more developers became Linux contributors. Huge corporate benefactors like IBM, Intel, and HP notwithstanding, Linux's popularity in devices, combined with its license, arguably created network effects leading to a truly massive numbers of contributors.

And it would appear in the rear-view mirror of history that Linus Torvalds, descended from Finland's Swedish-speaking "administrator class," was just the right bureaucrat, at just the right time, to benevolently govern all of that wild, creative computer programming energy. From the patches that made Linux portable, to Linux's rudimentary embedded filesystems (squashfs and cramfs), to the git tools for distributed (key word, there) source code management, Linus and friends always seems to come up with just what it takes to preserve and build on Linux's momentum.

Best Practice: When possible, choose components that are likely to benefit from the collaborative evolution resulting from open source licenses with stronger copy-left obligations (more onus on users to share their work back into the community).

But Wait... Share my OS... With My Competition? What?

It kind of seems like having to share everything with your competitors would be a bad thing, doesn't it? A reason to stay far, far away from Linux, and its "GPL contamination"?

For some, most definitely so. If your customers are buying your device because of the operating system, you may not wish to use Linux. Yet, how often do consumers really do that?

Most consumers do have an OS preference for a desktop computer, or a server. We're not talking about those, however, but rather about everything else.

For the vast majority of devices, no one cares about the OS. Do you care what your digital camera is running? Or your printer? No. You only care about the quality of the pictures. You probably didn't even know your digital camera had an operating system, or that it is very likely Linux.

In the device market, the OS is just middleware. The hardware may be a place to differentiate your product from your competitors'. The software may be a place to differentiate. All the OS does is glue the two together. Thus, it is an excellent candidate for standardization, or "commoditization" -- however you want to put it.

So, what has happened, is that for all intents and purposes, Linux has become the standard OS for device development.

As noted, every hardware vendor ships Linux. That includes both chip vendors and board vendors. They ship Linux "board support packages" (BSPs) as a convenience, so potential customers can easily 1). Ascertain that their product sample is functioning correctly, and 2) evaluate the performance of their application software on the product sample. Linux wins here because of its openness and popularity. Other than the work of bringing up Linux on the new hardware, which would have to be done anyway, there is no cost or penalty for redistributing Linux. And, essentially every computer science student is intimately familiar with compiling software for Linux.

Some vendors still ship BSPs for DOS, Windows CE, and VxWorks. However, these seem fewer and fewer all the time – though make no mistake, those platforms also enjoy widespread hardware support. Just not as widespread as Linux. That means you may not have the same consumer choice, in terms of peripheral parts, when building with non-Linux platforms. So, your cost might be higher, and your margins lower.

There are cases where a branded OS is appropriate. In mobile phones, consumers are becoming increasingly OS-aware, with most falling into the Apple or the Android camps these days. There may

be a few Blackberry holdouts remaining by the time this paper comes out.

Best Practice: Use an open source OS such as Linux if your consumer will never care or need to know about the OS brand.

Also consider using Linux when your OS will have a brand, but you wish to retain full control over all brand messaging, like the words on the screen, packaging, and advertisements. This is what Google did with its Android OS for mobile phones, for example.

On the other had, if you prefer to leverage the considerable marketing investments of third party OS providers like Microsoft, Wind River, Green Hills, QNX, and so on... then by all means, evaluate those products. They probably have not stayed in business as long as they have, in most cases, by selling stuff that doesn't work.

More Reasons To Like Linux

So far, we've looked at how Linux's fortunes were boosted by the right license at the right time, and by a virtual memory approach that simplified the creation of a truly reliable device. There was also the motivation of product companies to achieve better vertical integration and retain full branding control. Ancillary factors included the sudden availability of ridiculously cheap, MMUful and MMUless 32-bit hardware, and ever-cheaper memory and storage.

Believe it or not, that still isn't the end of the story, where Linux's rise to world domination is concerned.

An OS For The Day

Historically, there have been three technical advantages often cited as catalysts for Linux's explosion in devices. Over and over, device developers praised Linux for the following reasons:

#1 - Linux Rules the Net

Apple originally targeted designers in the desktop publishing market, and to this day, markets primarily on design aesthetics. Funny, it seems that design aesthetics appeal not only to designers, but also to everyone else. This previously unsuspected fact has made Apple the highest-valued publicly traded company ever. Go figure.

PCs evolved from the start with spreadsheets and office software, and continue to market on practicality and productivity, if not outright populism (check out the messaging of the Bill and Melinda Gates Foundation).

Linux, meanwhile....

Linux is literally **of, by, and for the Internet**. More than any other operating system, it evolved with the Internet, and as a result, has networking in its DNA.

If you've ever built a kernel, and enabled the advanced networking options, you know what I'm talking about. Linux is where a lot of cutting-edge networking technologies get developed, and the available options in there are mind-numbing.

How significant is networking to Linux? In the 3.2.9 kernel current at this writing, the networking stack and drivers account for fully 15 percent of all source code!

Besides its already-discussed strength in device drivers, Linux has protocol support for all but the very largest class of routers and switches. Or, for the simplest hand-held wifi-based IM gadget. And everything in between.

More recent open source projects like the excellent ConnMan are bringing Linux's network prowess further up the stack, too. Even the Network Mangler, er, Manager, is incredibly good nowadays.

Believe it or not, with many RTOSes, networking was once an add-on you paid extra for. With Linux, it is built right in, fully configurable, as simple or cutting-edge as you wish.

Linux's networking prowess played well in the early Otts, when suddenly, consumers expected to "connect" every device, be it a camera, printer, ipod, burglar alarm, you name it. Furthermore, consumers wanted to remotely configure everything via a browser-based interface. They didn't want to have to install something on their computer just to control a device.

Ironically, it was really Microsoft that was responsible for this. Experience with Windows left most people with the impression that the more software you install, the slower your computer gets. This may be true only of OSes with application registry databases. But nevertheless, in the early Otts, client-server was out, and the web apps were in in.

Linux benefits from at least ten different production-quality, embeddable open source webservers, many with CGI, lightweight scripting engines like Lua, and even AJAX support. If you need SSL, you can use full OpenSSL, or DropBear, or several other even smaller, configurable encryption libraries. Linux was just made for doing this kind of thing.

As a historical footnote, I seem to recall today's bling-rich AJAX web techniques sprouting from some pretty humble origins: remote sensing geeks looking to save solar power. With Ajax, rather than just responding to client requests, the server itself could initiate communication. So, it could just sleep until something urgent awakened it, rather than continually having to power up just to respond to remote polling. Awakening and going to sleep again represent the vast majority of the power budget in many systems of this type, so it was a pretty big advance. But AJAX was great for other low-powered devices, too, because the more the processing burden could be shifted from server to client, via Javascript, the more responsive the user interface could be made to feel.

Best Practice: If your device is connected (what device isn't, these days?) consider an OS such as Linux that will offer you a lot of options, both for configuration and driver support. Nearly universal hardware support may increase your purchasing power, letting you source the cheapest parts for each production run.

#2 - LINUX HAS MULTIPLE DISPLAY OPTIONS

With Linux, when it comes to the display, you have options. For the tiniest system, you might just put the console on a serial port. For deeply embedded systems, there's a kernel framebuffer that abstracts the video hardware... letting you write even very minimalist GUI layers without marrying any one specific hardware provider. There's even hardware acceleration, via directfb.

Moving up in functionality, there's SVGALib, which has at times been popular with gaming software authors. Simple Direct Layer (SDL) is another step up that has been trending upward in recent years. Finally, if you need the utmost in graphic performance, you can just use Xorg; that opens the door to doing everything on a device that you can do on a PC.

This kind of flexibility helps Linux suit a variety of product types, and is especially useful where a line of products must span a range of price points and functionalities.

Best Practice: Along with the processor, the display is the most expensive component in a device. Consider an OS such as Linux that will give you some options, including choosing the lowest-cost parts, and in some cases addressing a wide range of price points with a great deal of shared code.

#3 - Linux PLAYS MULTI-MEDIA WELL

Multimedia performance is another area where device developers constantly praise Linux:

- The kernel itself has excellent throughput.
- The kernel is highly configurable for multimedia quality-of-service.
- There are several codec frameworks to pick from, like gstreamer, helix, and mencoder. People say GStreamer has made huge performance strides in the most recent releases.
- There are many player options, like mplayer, xine, totem, alsaplayer, aplay, mpc, to name only a few.
- There are several driver frameworks... OSS, Alsa, and the relatively new OSS4 option, among others.
- There are various audio routing layers (esd, PulseAudio and Jack), and the latter two both have pretty darned good real-time performance.

Best Practice: If your device needs to render multimedia, especially on low-powered hardware, you may wish to evaluate Linux.

But Wait - Linux Must Have Some Flaws Too

Oh, you bet. And you can probably guess what the biggie is.

There's this community, in your product, serving its own agenda.

Working With The Embedded Linux Community

Product developers have one set of goals, that can usually be stated as: a relatively stable, flexible, configurable platform with universal hardware support, and a universe of freely reusable, royalty-free software to choose from. This can get them to market on time, and on budget.

Open source developers have a vastly different goal: to create the very best thing of its kind. Attaining that goal will bring users, massive testing, and in attract more and better contributors. That in turn leads to "network effects," and ultimately to commercial licensing and service consulting opportunities.

Even better, being the best at something, or creating the best thing of its kind is just undeniably fun, for those possessed of the engineering mindset.

A Divergence of Purpose

So product companies want one thing – stability and long product life-cycles -- and open source developers (including Linux developers) want rapid progress. When these goals collide, embedded product companies may feel like they've hitched their wagon to a rocket-ship.

Sure, you get to take advantage of all the new "features" and performance and especially new hardware support that arrive with each (micro-)evolutionary step. But instead of just getting a new release from your RTOS vendor every other year, with Linux, there is a steady flow of releases you have to decide how to manage.

Due to the desire of developers to make Linux the best of its kind, the "application programming interface" (API) undergoes perpetual, shall we say, "evolution." It's hard to think of a single Linux subsystem -- be it sound or video or wireless networking or USB or peripheral interface detection and naming or scheduling or... you get the idea – that has not gone through at least one or two major rewrites in the last decade -- let alone over the whole course of Linux's 20-plus years.

Every time Linux's APIs change, you have to change your code, too. So, maintaining your applications on Linux may require more work than on less dynamic OSes. It's a lot of work to keep

up – yet doing so is the best way to ensure good security, and to be able to get support from kernel.org. Linux hackers may not be interested in fixing a bug in some very old version that few people use any longer. But, if you find a fault or regression in the current release, they will trip over themselves to solve the problem!

Best Practice: Using the newest releases of any open source software gives you the best chance of getting help from the people who know the code best – the authors themselves! And, since most people try to keep fairly current, you may also get more help from those who, like you, are implementing it.

Managing Up

Many open source adopters find that pure, pristine upstream code comes very close to fitting their needs – yet may fall short in a few areas. It is no different with the Linux kernel. Luckily, the source is available, so with a few hacks here and there, your product requirements may all be met. The downside is that such "hacks" will have to be ported forward for the foreseeable future.

There is another possibility. If you are willing and able to work with upstream developers, you might be able to get your hacks merged. If so, they will likely be maintained by others.

If you're talking about a simple bug fix, it's definitely worth a try. Just submit it upstream. If it's a new feature, the sell may be tougher. However, in recent years, the Linux kernel team has been pretty gung-ho to accept patches from embedded developers.

People tend to blame – or credit – Andrew Morton. Around 2007, the noted Linux Maintainer began stumping for increased embedded participation. See http://goo.gl/Yo7OI and http://goo.gl/OU9gT.

Morton's talks should be required reading for those considering submitting patches or features to kernel.org. He's full of great suggestions, like (these are paraphrased):

- It's okay to use commercial Linux in consumer products. But prefer generic kernel.org where the customer may reasonably be expected to upgrade the OS themselves
- If your team and/or company is going to work with kernel.org, designate a single point of contact
- Working with kernel.org will make your code better, because it'll be massively peer-reviewed by kernel.org developers, and massively tested

For the best chance of being taken seriously, be sure to follow the knit-picky coding style guide delineated in the kernel sources, or here: http://www.kernel.org/doc/Documentation/CodingStyle . If you don't like your tabs eight spaces wide (maybe you're on a laptop) try setting your editor's tab width to something else, and then using "indent" to reformat prior to submitting.

Also, before joining the lkml mailing list, don your "asbestos undies," as the culture favors merit over civility, for sure. In fact...

Best Practice: Managing your non-differentiating patches upstream into the Linux kernel itself is often a great idea. However, it is not something to be taken lightly. Go in prepared for what's ahead. A good place to start might be Jon Corbett's paper.

And come to think of it, now would be a pretty good time to drop the big one:

Open Source Best Practice Number One: Stop Coding

For anyone wishing to leverage open source code, whether in an embedded Linux device or just a simple website, here is a great pearl of wisdom: It might seem self-evident, but the first step in leveraging open source is actually to stop writing code!

If you write something, you are probably going to have to maintain it. Whether it be a simple patch, or a complete application, you're taking on an ongoing responsibility, like a monthly recurring cable TV charge. Each time you wish to upgrade the open source parts of your stack, you will have to first port your patches forward to match the new release. That can involve a lot of relatively uninspiring work.

Much better to stick to non-recurring charges, if you can. Use your time to incorporate pristine upstream code that is likely to be maintained by someone else. And that, arguably, was the great lesson commercial Linux vendors taught everyone in the mid-Otts: "Stop carrying all those patches!"

Binary Blues

One thing that is unusual about Linux, among operating systems, is that it comes with absolutely no guarantee of backward binary application interface (ABI) compatibility. That means you cannot count on binaries built for one kernel working on the next, no matter how minor the patch level. They might run fine. Or they might not.

This results from a design philosophy aimed an letting Linux evolve quickly, rather than being tied to possibly inefficient methods from the past. Thus, in terms of meeting the goals of open source programmers – to build the best thing of its kind – the lack of backwards ABI compatibility is a great thing that truly frees Linux developers to innovate free from inefficient encumbrances from the past.

Meanwhile, many chip companies do not publish the source code, due to legal fears. Instead, they issue binary-only drivers from time to time, typically built against the kernel versions adopted by the largest Linux distributors, e.g., the commercial embedded Linux distributions and the big desktop Linux distros such as Red Hat and Ubuntu.

So, the problem is clear. You may wish to upgrade to a newer Linux kernel, for the reasons cited above. However, you may have some drivers that are tied to an older kernel build. Your options, then, are limited. You can try to back-port the features you need from the newer kernel. Or, you can approach your hardware supplier(s) and or your suppliers other customers in hopes of getting your hands on a driver built for a newer kernel release.

This situation is a real set-back, reminiscent of the days when device builders relied on an RTOS supplier for drivers. Yet, it is the reality "on the ground" for many device engineers. If you have multiple binary drivers, the situation is compounded. A newer build of one may be available, but unless you can get the newer one upgraded as well, you may still be stuck with the old release. Or, perhaps you will have to spend some time reverse-engineering the binary driver, and writing a wrapper for it, so it can use the newer ABI.

Best Practice: When possible, try to minimize the number of binary drivers in your stack. When not possible, try to engage with the community to see how others handle the lack of current drivers.

Toolchains, And Other Oft-Forgotten Working Class Heroes

C is the language of the Linux kernel, so you need a C toolchain to build a Linux kernel. You also need C tools to build your tools, your C libraries, and ultimately your filesystem.

Confused? Let's back up. In embedded-speak, the word "filesystem" has several meanings, one of which amounts to: the rest of the software stack, kernel notwithstanding.

The filesystem that ships in a device would rarely include a toolchain, since tools are typically needed to build but not to run the operating system. However, the toolchain is a truly important part of the filesystem on your development system.

"Toolchain" can mean almost anything. At a minimum, it comprises a compiler and linker. The

compiler turns human-readable code into machine-readable "binary" (1s and 0s) "object" code. Building even a simple application may produce quite a few binary objects, because C code typically "includes" external resources, like the C libraries. That's where the linker comes in.

The linker's job is to put objects together so they can find and work with one another. The linker can "statically link" everything into a single, big, self-contained binary object. Or, it can create an object capable of "dynamically linking" pre-built objects elsewhere on the filesystem. The location of such objects is typically specified at configuration time.

On Windows, pre-built objects are known as DLLs, or "dynamic-link libraries." On Linux, they are called "shared objects." The good ol' GNU C libraries supply a good many of the shared object files on a typical Linux box: 291, it appears, on my fairly recent Ubuntu box.

Best Practice: Dynamic linking works best when the same toolchain that is building your app was also used to build any shared objects linked by your app. Since the GNU C libraries comprise the most commonly linked objects on most Linux systems, it probably makes good sense to upgrade your toolchain and C libraries together.

Tool Selection and Upgrade Cycles

Toolchains are a somewhat esoteric aspect of programming. They are deeply understood by relatively few people. Built and maintained by specialists, they are often supported by both product and service companies.

Some of the complexity stems from the fact that in the device world, most toolchains are hosted on one architecture (typically x86) but "target" another, like ARM, MIPS, SH4, or PowerPC, or whatever. These are called "cross-platform development tools," or commonly just "crosstools."

Crosstools-ng is the "old standby." It can build crosstools for uClibc, a really tiny C lib for MMU-less and/or deeply embedded processors. Or, it can build tools against glibc, the normal GNU/Linux C library. And, it also supports eglibc, a relatively new C library aiming to be full-featured, yet small enough for embedded use. Crosstools-ng only builds toolchains, however; it does not aim to be useful for anything beyond that.

Another, newer option is buildroot. It grew out of the uClinux project, but like crosstools-ng, can also be used to create cross-platform toolchains based on uClibc, glibc, or eglibc. It goes further, in that besides just building toolchains, it can be used to build a complete filesystem, such as a ramdisk image.

Both crosstools-ng and buildroot are fairly easy for a complete novice to get working. Both are typically introduced to entry-level embedded Linux students – for example, in embedded Linux classes offered by the Linux Foundation.

Like the Linux kernel, busybox, and much other embedded-specific software, both use the familiar "Kconfig" or "make menuconfig" configuration script. Kconfig is familiar, and fairly easy to customize. Those using buildroot to produce device filesystems probably get pretty comfortable with adding configuration options and help texts to its hierarchical configuration interface.

Best Practice: Would-be embedded Linux engineers would do well to learn Kconfig, as it would be difficult to build any kind of embedded Linux device without encountering Kconfig at one or more points.

An Aside: The Orthogonal View

Buildroot sometimes wins praise for its forbearance from feature-bloat. It gets a lot done, but aims to do so by leveraging pure, upstream versions of other open source projects – such as Kconfig. This epitomizes one of the core early design philosophies behind Unix: orthogonality.

Dennis Ritchey (may he RIP) believed tools should be "orthogonal," like the separate 2D views in a mechanical drawing. Each "perfects me first" (oh sorry, that's Dr. Bronner again. How does he keep getting in here?). Each does one thing, and does it well. Yet, each combines easily with other tools, for example, through Unix pipelines.

At the risk of stretching the "mechanical drawing" example too far, any three adjacent and equally spaced 2D views in an orthogonal drawing can be extended, at 0, 45, and 90 degrees respectively, to create an accurate 3D projection.

Watch Out For Really Esoteric Toolchains

A great majority of Linux devices are built with tools and C libraries from the GNU project. However, there are many other compiler options to choose from.

Weird hardware is pretty common in the device market. Architecture vendors like ARM and MIPs and their partners are forever pushing little bits and bobs that promise the "order of magnitude" performance gains and power drops that come from doing things in hardware rather than software.

Sometimes, the resulting hardware requires compiler support. Customers of such hardware may do well to consider ecosystem health, however, before committing to an odd-ball toolchain. Toolchains are tied to specific C library releases, so infrequently updated tools could impede your ability to keep pace with the speed of open source advancements.

Upgrading a toolchain is not trivial. Sometimes likened to a "black art," it is a multi-step process in which the compiler is slowly, painstakingly assembled until complete enough to build itself.

Best Practice: If relatively few people understand how to build your toolchain, or if you think your toolchain may not be upgraded often, it may prove beneficial to invest in some in-house toolchain expertise.

Distro Vs. Filesystem Maintenance

Smaller product development teams tend to compile firmware stacks for each product release, using tools such as crosstools-ng or buildroot. Larger companies, though, may employ a developer or even a team to maintain a complete in-house distribution. Individual product development efforts can then get a jumpstart by tapping the company distro.

The last few years have seen much interest in creating tools to help companies create and maintain in-house embedded Linux distributions. OpenEmbedded (OE) was one of the first highly successful efforts to emerge, in this area. OE resembles Gentoo Linux, in that its package management tools pull source code, either from upstream or local sources, and compile it on the fly.

OpenEmbedded's build process is based on Bitbake "recipes." These have a reputation for great flexibility, at the cost of a considerable learning curve. As a result, commercial OE support has become a cornerstone for several embedded Linux service companies (Embedded Alley comes to mind).

Meanwhile, several downstream embedded Linux distributions that use OE have gained large user bases, including Angstrom, KaeilOS, and Openmoko. Palm/HP's commercially ill-fated, but now open source WebOS distro was also maintained in an OE environment.

The next evolution for distro-maintaining tools was Yocto, which emerged in 2010. Yocto appears to enjoy considerable support from the Linux Foundation and its partners. It "aligned" itself with

OpenEmbedded in March, 2011.

Compared to OE, Yocto aims a little higher in the stack. Besides just the tools, the project is taking ownership of quite a few actual bitbake recipes, for many common architectures. Besides making OE easier to use, this could help reduce fragmentation, Yocto proponents say.

Best Practice: If your company creates many products with shared hardware, interfaces, or software features, standardizing on an in-house distribution makes sense. Specialized tools like OE/Yocto stand to offer considerable convenience, as well as conformance with industry norms in library and tool selections. There is always risk in basing an in-house distribution on an external project like Yocto. However, Yocto appears to enjoy substantial backing.

Give Linux The Boot

Developers accustomed to x86 may not give too much thought to the bootloader, which awakens and initializes hardware, nor to the bootstrap loader, which finds, uncompresses, and hands over control to the operating system kernel. All PCs follow certain "BIOS" standards, so that any compliant hardware automatically gets found and initialized, and the same for any compliant OS.

With embedded architectures, there's less standardization (to say the least). Especially in the ARM architecture, system-on-chip hardware may be assembled from an unpredictable mix of IP blocks licensed from ARM, LTD, and its partners. There seems to be little address space standardization among SoC vendors, or even between different SoCs from the same vendor. As a result, the capabilities of the bootloader shouldn't be taken for granted.

Best Practice: When evaluating Linux on a particular hardware system, the capabilities of available bootloaders should be evaluated as well.

Filesystems

The Linux kernel supports many filesystems, including some intended specially for use on the "flash memory" typically used in embedded devices.

Flash was named for the way camera flash bulbs work: when it's time to erase flash – to turn the little 0s back into 1s – a great deal of electrical energy must be stored up and then released in a burst, in order to overcome static pressure. As you might expect, this technique results in the eradication of a fairly large number of 0s – in fact, a full block's worth.

Modern NAND flash device blocks might be 256K or even larger. That means in order to change one character in a log file, you have to first copy 256K (with the changed character), and then blast away the stale data. All this bursting and blasting takes its toll, and a given "erase block" can endure being flashed only so many times.

To prevent flash from wearing out quickly, many flash filesystems attempt to "wear-level"; that is, to spread erasures evenly among all the blocks. The traditional figure cited was 100,000 erase cycles. That seems to have increased by an order of magnitude or more, at least in marketing literature.

Flash can be loosely divided between NOR or NAND approaches, named for the respective bitwise techniques used to read data. NOR is the old, expensive, relatively reliable kind used mainly in industrial applications. NAND is cheaper, denser, and typically found in consumer devices.

Traditional "embedded systems" based on Linux typically used a small amount of expensive NOR flash to store critical things, like the OS. Where larger capacities were needed, some cheaper NAND would also be present. Both types would typically connect as "raw" memory devices; that is, any wear-leveling would have to be handled by the Linux kernel and associated filesystems.

A few of Linux's earliest flash filesystems, CramFS and SquishFS, do not bother with wear-leveling.

They do not really have to, since they are essentially “read-only.” Written by Linus during his days working for chipmaker Transmeta, CramFS and SquishFS essentially just shake out files at boot time and fluff them up onto a RAMdisk – the difference being in how tightly the files are serialized when stored.

An aside: Raw flash devices are not to be confused with flash memory gadgets like USB keys, SD cards, CompactFlash, and the like. Such devices have an on-board memory controller, typically running on an ARM7 core, that handles wear-leveling being the scenes. Such devices attach as normal block devices, and act as hard drives (albeit slow ones). This approach was invented by M-Systems, which marketed the first such devices as “disk-on-chips.”

The Linux kernel supports several flash filesystems that do handle wear leveling for Flash devices connected via Linux’s MTD (“Memory Technology Device”) interface. They can be found under “Miscellaneous Filesystems” in menuconfig. Through the years, each has waxed and waned in popularity: JFFS2, LogFS, and UBIFS are those present in the current kernel.

JFFS2 has long been popular for fairly small flash partitions. LogFS is said to suit really large flash devices; however, it is sometimes described as “a journal without a filesystem,” so depending on your application, it could use more processing power. Anecdotally, UBIFS seems to be chalking up many design wins.

Another mature option is YAFFS2, which is maintained outside the kernel tree, and also sees use with other RTOSes. It has been around for a long time, and has seen many design wins.

Best Practice: Many device developers benchmark available flash filesystems with their specific application. And, considerable work often goes into tuning filesystem parameters, in order to achieve the greatest performance and reliability.

Increasingly, high-end devices like smartphones are using a relatively new storage technology known as “eMMC.” Along with copious volumes of high-density NAND storage, eMMCs employ their own on-board memory management controller (MMC). The MMC lets the eMMC connect to Linux as a block device, similar to a USB key, SD card, or hard drive. The MMC uses a particular flash memory vendor’s preferred, proprietary wear-leveling techniques. Typically, you would format your eMMC device with a normal block filesystem, like ext4.

On the one hand, as a developer, this is great, because the complexities of flash wear-leveling are wholly abstracted and cloaked by the eMMC. On the other hand, tuning your system for optimal stability and performance may involve trial and error, as your storage device is literally and figuratively a “black box”.

Yet, eMMCs seem to be where portions of the device industry – smartphones, for example – are heading.

At Last - The Filesystem

Wait, didn’t we just discuss the filesystem? No silly, not that filesystem! I’m talking about all the rest of the software in your stack. Where are you going to get your Linux???

The time-honored tradition has been to use source code packages from a popular distribution, like Red Hat or Ubuntu. That way, you benefit from the additional testing these well-maintained versions receive.

Naturally, this fact led someone to see an opportunity, and today, a startup called Linaro has jumped with both feet into the business of packaging up Ubuntu, the most popular distro, for embedded architectures (mainly ARM, I think). This seems like great news, and besides that, Linaro seems bent on bringing some sanity to the crazy world of bootloaders and “architecture” files for all

the zillions of ARM-based SoCs out there. Many in the embedded Linux world no doubt wish this company well!

Once you decide on your source, you still have to pick packages. You may also at times have to bring in extra-curricular packages – "Universe" items, in the language of Ubuntu. Every time you bring something into your stack, it may be worth considering its long-term viability, roadmap, and ultimately, its alignment (or not) with the goals of your project.

Close Editor, Open Browser

Time to hit the mailing lists, evaluate website traffic levels, and ask around. Maybe download some stacks to see what others are using. The aim is to evaluate the long-term prospects, vitality, direction, competence, and general momentum of each alternative, and then pick the one most closely aligned with your goals.

To be honest, this "evaluation" step can amount to more work than just writing something yourself. The payoff comes down the road. See the part about "other people doing the maintenance." Oh, and the fact that a lot of other people will help you test it.

We covered this already, with regard to kernel hacking, but it bears repeating in the context of the rest of the stack, too: If all you do is code all the time, you'll miss out on the true benefit of open source. You'll continually re-invent the wheel, rather than innovating on top of what already exists. And, your work will not be as widely reviewed, or tested, or appraised. It will almost certainly be inferior, as a result, to alternatives you might have used for free, had you only had the patience to look around a little but longer.

Determining what components to include and which to build on is boring, administrative work. On the upside, the more of this type of thing you do, the more familiar you get with the landscape, and the easier it gets.

As you get better and better at it, you may find yourself ripping out more and more things you wrote yourself, and patching in code that was written and is being maintained elsewhere. You may also find yourself deleting functions you wrote, in favor of methods that were there in your library – or in Linux – all along.

So ultimately, to apply open source well, you may wind up doing more reading than writing. More listening than talking, if you like. And while your job description changes, it should ultimately get easier.

License Tracking

Quite a bit of open source software is available under multiple licenses. BSD, MIT, GPL, take your pick. That can simplify things a great deal. However, it is not enough to just look at the root-level project license, and assume it applies to all the code in the tree above.

Where things really get complex is when open source software projects themselves draw from multiple other open source projects. As a result, the code in various subdirectories typically will contain additional license stipulations. Just collecting all the licenses in once place is a serious challenge!

And that challenge, it turns out, is one where developers can make a huge difference. More than anyone else, developers are in the source code. Their job should not be to try to interpret the law. But increasingly, developers using open source may be tasked with creating a license manifest – basically a list of all the licenses they run across.

If you're lucky, your company may have a legal team to help evaluate licenses. Often as not, though,

at least some part of open source license evaluation falls to developers, according to surveys at LinuxDevices.com as recent as 2007.

The challenge posed by open source license management led to the launching of several startups, including Black Duck and Palamida. Subsequently, open source packages like HP's well-publicized "FOSSology" came along. Of course, each took slightly different technical approaches, and each devised slightly different document formats.

One ray of hope, albeit still a very young one, is the new SPDX standard. Among other promises, it could provide some standard formats and approaches for open source license management.

Conclusion

So, there you have it. Linux achieved world domination in devices thanks to technical strengths, good management, a great license, portability, source availability, royalty-free marketing, product and service support from commercial partners, and a host of other factors. While embedding Linux is not without challenges, at this very instant, there are probably more running instances of Linux than any other operating system. And with the rate of development that Linux continues to enjoy, it's only going to get better.

About the Author

Henry Kingman observed and wrote about embedded Linux markets and technologies for the better part of a decade: first as a contributor to LinuxDevices in the late 1990s, then as that site's chief writer and later editor from 2003 to 2009.

Why Train With The Linux Foundation

The Linux Foundation offers several embedded Linux training courses:

Embedded Linux Development (LF411)

Get advanced Linux training on the key steps to developing an embedded Linux product. Gain real world experience through extensive hands-on practice with target devices.

Building Embedded Linux With The Yocto Project (LF405)

Gain a solid understanding of embedded development using the Yocto Project, including the Poky build process and Bitbake, the use of emulators, building images for multiple architectures and the creation of board support packages (BSP).

Introduction to Embedded Android Development (LF308)

This class will teach you how the Android build system works and how to add a completely new device definition, how to customise the components that go into the build, how to obtain and build a Linux kernel with Android additions and how to load it onto the new target board and configure the boot process.

Additional Developer Training Courses

The Linux Foundation offers a full selection of Linux training courses for developers, including courses that focus on Device Drivers, Kernel Internals & Debugging, and Developing Applications. See a complete list of developer courses at training.linuxfoundation.org.

Distribution-Flexible

The Linux Foundation's courses are built to be distribution-flexible, allowing companies or students to easily use any of the big three distribution families: Red Hat/Fedora, Ubuntu/Debian, SUSE/OpenSUSE. If your company runs one of these Linux distributions and needs an instructor who can speak deeply on it, we have a Linux expert who knows your distribution well and is comfortable using it as the basis for any corporate Linux training. For our open enrollment students who take our online training or classroom training, our goal is to help them, first and foremost, to become Linux professionals, rather than focusing on how to use one particular set of tools.

Technically-Advanced

The Linux Foundation's training program has a clear advantage. As the company that employs Linux founder Linus Torvalds, we are fortunate in our ability to leverage close relationships with many of the top members of the Linux community, including Linux kernel maintainers. This led to the most comprehensive Linux training on the market, delivered through rigorous five-day courses taught by Linux experts who bring their real world experiences to every class.

Since Linux is always evolving, our course materials are regularly refreshed and up-to-date with stable versions of the Linux kernel. We deliver our advanced Linux training in a 50/50 training format, where 50 percent of a student's time is spent learning from an instructor and the other 50 percent doing exercises in hands-on learning labs.

For more information about our Linux training, please visit training.linuxfoundation.org and contact us today.

LINUX FOUNDATION

The Linux Foundation promotes, protects and standardizes Linux by providing unified resources and services needed for open source to successfully compete with closed platforms.

To learn more about our Linux Training program, please visit us at training.linuxfoundation.org.