Designing Your First Embedded Linux Device 3

Choosing Your Software Stack

Nathan Collins | Senior Software Engineer, KDAB

KDAB

Deciding on the various software components in your stack is a crucial step when creating your first embedded Linux device. You want to build a stack that meets your objectives now and brings continued value in the future. However, there are so many tools you can combine to build and maintain a successful product, it can be difficult to know where to start.

This whitepaper looks at the choices you have for the entire software stack from the OS to the application and gives you things to consider at every step.

Operating system concerns

Should we go with Linux or something else?

Embedded Linux is the right choice for most, but not all. Linux is pretty good at responding to devices within a reasonable time – but it makes no guarantees. If your product has enough time-critical needs for responding to certain stimuli, you might want to consider an RTOS such as QNX Neutrino RTOS, GreenHills INTEGRITY, or VxWorks.

These operating systems allow you deterministic control in managing time-critical devices. They are all POSIX compliant, so while they aren't technically Linux and will have some specialized commands, tools, and APIs, in general, they have a Linux-like feel and should compile most Linux source without trouble.

From experience: The costs of a swipe

iPhone user interfaces not only need powerful graphics, but responsive touch screens. We worked with one company who was trying to "iPhone-ize" their product on a very limited processor that was not much more than a microcontroller. We helped them cut their original design to better fit the hardware by eliminating animations, gradients, and other flourishes. But also problematic were the touch-screen gestures. Because no physical buttons were planned for the hardware platform, the UI couldn't fully eliminate gestures. A lot of effort was expended in optimizing and fine tuning to disguise how non-fluid the graphics were. Mobile phones have set consumer expectations very high, and companies can spend a lot of time and money trying to meet those expectations.

Time-critical Linux

Linux isn't as intimately controllable in the time domain as an RTOS, and it won't ever be hard real-time. However, if you want to stick with Linux for your mainline development, you have other options besides an RTOS for handling time-critical tasks. For instance, you might be able to get away with a real-time patch to the kernel, as long as you avoid other known trouble areas like page swapping.

Rather than hacking Linux into a task it was never designed to handle, it's probably better to give your real-time tasks the regular servicing they require with a little outside assistance. (Some of these solutions can also work well if you have power-sipping modes so you may be able to solve two problems with one architecture.)

- **Hypervisor.** You can add a hypervisor that sits underneath the OS. This allows you to run some code alongside Linux – either via an RTOS or bare metal code that can handle the timing of important tasks.

- **Onboard processing.** Many CPUs have companion processors on the die such as an ARM M4 or DSP. If these co-processors have access to the needed hardware and aren't required for other tasks, you may be able to use them for time-critical purposes or leave them on when the main CPU is powered down.

- **Microcontroller.** Dedicating a separate microcontroller can be a good approach for time-critical or low-power tasks. These can be very simple processors with no need for a sophisticated OS. The software is also simple – just an event loop to poll your devices or a perpetual sleep mode that handles anything needed via interrupts.

- **Dedicated core.** Although this can be a huge waste of processing capacity, you may be able to reserve a core for your time-critical tasks. This approach can also work for low-power applications since some CPUs have cores that are tuned for either performance or power consumption, letting you shut off the higher-current cores when they aren't needed. This requires some special configurations to manage and necessitates page locking the real-time code and data, but it might be a reasonable approach if there are no good alternatives and you've got processing power to spare.

In all cases, whether a solution works properly or not is completely dependent on the demands of your time-critical hardware and how you need to access it. As an example, if the hardware you need to service is on a bus that's only accessible by the main CPU, a microcontroller or co-processor solution is impossible.

No matter what the solution, you're adding complexity to the design. You need two software environments and toolkits. You need to communicate between time-critical and non-time-critical software. This very important channel must be designed very carefully so as not to become a bottleneck.

From experience: Browsers need big horsepower

HTML has a big advantage in cross-platform development, as well as being able to naturally support remote UIs. However, browsers use many layers of abstraction and software to create an interface. We worked with one customer creating a SCADA system who had designed a browser-based UI. They used Qt Web Engine, which has GPU acceleration due to its Chromium roots – normally a good choice, but it's designed with faster hardware in mind. In this case, the hardware rendering on this older GPU was slowing down the browser. We were able to double the frame rate by avoiding the GPU accelerated code path and switching to software rendering. Unfortunately, the frame rate was still slow at under 10 frames per second and there was little else that could be done using this browser environment and fixed hardware combination.

Fine tuning the filesystem

Most users don't ever think about the filesystem. However, if you're building an embedded system, it can be something you think about a lot. Here's a few of the things we think about – the three 'W's and three 'H's.

- **Where.** Should you put your file system on board the chip or on a removable storage like an SD card or USB flash? Whether you need to do this depends on the size of the onboard storage; if it's not big enough, you'll have to augment it with something you can plug in. That's also the case if you need quick replacements either for instant firmware upgrades, in-field diagnostics, or other large and instant data transfers.

- **Which.** Linux offers a wide variety of file systems types that you can choose, such as ext4, JFS, ReiserFS, XFS, and Btrfs. These all have a wide array of benefits and limitations to consider, such as maximum number of files, maximum file size, file name length and encoding, file system overhead for many small files, file naming conventions, commonality/exchangeability, compression, performance, automatic leveling, fragmentation behaviour, speed on starting up, concurrency behavior, and crash resilience.

- **Writeable.** Does your filesystem need to be writeable? In most cases, yes but maybe not the whole thing. Making one volume writeable for logs and user data while another is read-only can prevent corruption or cybersecurity issues, but because most applications can develop intricate file system dependencies, this is something that you should consider early in application design.

- **How fast.** Does your file system need to be as fast as possible? If you're constantly streaming large amounts of data for video recording or other sensor collection, you want to choose the flash so that the hardware can keep up with read and write demands and choose the filesystem so that it works well with continuous writing of large blocks.

- **How much.** Are you writing and flushing constantly to the file system for persistent logs or database updates? Not only does this have performance considerations, but it can also prematurely wear out your flash storage. Most flash chip controllers have built-in wear leveling, but you should still perform some back-of-the-napkin estimates to see how long your flash chip can be expected to last. As the disk space on your product's flash fills up, will you have enough space to add new features? Also critically, will you have enough space for your update mechanism (if it needs to swap firmware images to update)?

- **How long.** If your system is expected to last a very long time in use, the lifespan of the flash contents becomes an issue. Most flash storage has a maximum length of time it can store bits before they start to decay. While that's usually in the order of years, if your product is designed to withstand sitting unused for long periods, this can be a problem. Used constantly, the flash can rewrite to refresh bits before they decay, but infrequent use can shorten your product's lifespan.

From experience: Flash speed seriously impacts boot speed

There is a very large range in performance for mass storage flash. In many cases, normal system behavior doesn't demand anything particularly speedy. But slow flash chips are not compatible with a quick booting requirement – and these are often the ones that cannot be upgraded since they are soldered in place. We've seen many customers struggle with this when they're assuming that flash size is their biggest issue when it turns out their larger problem is the speed of their flash, which significantly limits how quickly they can boot the system. Automotive is one industry where boot speed is a well-known concern, but consumer products too often need fast booting. An "instant on" approach lets the user get long lives out of their limited battery – but only if there isn't a perceptible time difference between a cold and warm boot.

Managing the distribution and board support package

Presumably, you're doing the lowest-effort approach by getting a Linux distribution from your board vendor. If that's the case, you need to answer a few questions about that distribution and its board support package (BSP). The biggest of these: is your vendor's Linux suitable for use in production? Not every board vendor wants their Linux used for real customer systems and that's especially true for silicon vendors who almost always expect their Linux is for eval purposes only. They don't want to be on the hook for any bugs or unapplied patches and that can put you back in the OS business.

Many board vendors now let you build your own Linux – and that's a great thing. Yocto and Buildroot are probably the two most popular options that let you easily create and maintain your own distribution. Even if your Linux is not supported by the board vendor, this is probably the best approach since customizing the distro is so much easier with a tool rather than by hand. Like anything, there are pros and cons to each tool, but whichever one you pick, it's a great way to approach your first embedded Linux project.

Planning the application

We can't tell you how to build your application, but we can tell you some things to consider when planning out the software.

Microservice or monolithic

Is your application better suited to a monolithic application or multiple services? Microservices compartmentalize an application into many small, individual components that can be developed, tested, and updated by independent teams. By loosely coupling the system functions together, they provide a lot of advantages in access, maintenance, testing, and delivery – among others. However, micoservices do require some overhead to implement so they may not be appropriate if you have a small development staff or if product decomposition doesn't lend itself well to isolated components.

Multi-user

Is your product used by multiple users? It can be a huge problem if you've created a single-user product that down the road needs to be converted to handle multiple users. Not only would this move require the addition of logins and authentication, but it would also create the need to manage all databases, logs, configuration files, and more on a per-user basis.

It’s far better to decide up front that you need multi-user capability and later restrict it to a single user than to go the other way. (We know, because like many of the challenges in this eBook, we’ve helped customers work through this issue.)

Containers

Should you containerize your application? Containers are easily adopted in a microservices architecture and provide many benefits like better cybersecurity, streamlined development, and simplified testing. They introduce some overhead – but not as much as you might think – and make for really simple development and production updates, so they’re worth investigating.

Security

Is the vendor-supplied Linux secure? In other words, is it being kept up to date, with all libraries and executables patched to remove any known common vulnerabilities and exposures (CVEs), and are they measuring security in any way that helps your team make risk assessments? If you’re using Yocto, you can find out yourself if there are CVEs in your Linux. To help with security, only keep the components you directly need – the more you use, the more you maintain.

Also important is to keep in mind things that can impact overall application architecture and that should be designed in from the beginning like eliminating root access, using read-only file systems (as discussed above), or running code in protected containers and sandboxes. If your device isn’t part of the IoT and doesn’t need Internet access or you don’t plan on updating it remotely via OTA, consider restricting network access to field diagnostics only or removing it altogether.

Summary

Building your first Embedded Linux device is not easy. Hopefully this guide gives you a good feel for the many things you need to consider. Our engineers have deep expertise in all aspects of embedded product development so please don’t be shy to reach out if you have any questions or need help at any point in the process.

This is the third whitepaper in a series of four that covers planning considerations and lessons learned in building embedded devices with Linux. Each whitepaper addresses a specific portion of the development lifecycle, so you can easily focus on the guide most relevant to your current stage of development. If you don't find the advice you need in this whitepaper, check out our first, second, and/or fourth whitepaper in the series.

View the four parts of this whitepaper online:
www.kdab.com/publications/embeddedlinux/

About the KDAB Group

The KDAB Group is the world's leading software consultancy for architecture, development and design of Qt, C++ and OpenGL applications across desktop, embedded and mobile platforms. KDAB is the biggest independent contributor to Qt and is the world's first ISO 9001 certified Qt consulting and development company. Our experts build run-times, mix native and web technologies, solve hardware stack performance issues and porting problems for hundreds of customers, many among the Fortune 500. KDAB's tools and extensive experience in creating, debugging, profiling and porting complex applications help developers worldwide to deliver successful projects. KDAB's trainers, all full-time developers, provide market leading, hands-on, training for Qt, OpenGL and modern C++ in multiple languages.

www.kdab.com