Designing Your First Embedded Linux Device 1

Framing the Development Process

Nathan Collins | Senior Software Engineer, KDAB

If your company is building its first embedded Linux device, you're going from a relatively easy-to-understand product environment to one that's software-dependent with thousands of technical decisions to make. Nobody wants the expensive mistake of a failed product, but without previous experience, how do you go about creating an embedded system that is not only successful with customers but also a solid foundation for future innovation?

This is the first module in a whitepaper series on designing your first embedded device; it covers the beginning and ending of the product development process. At the onset of a new project are a handful of critical choices that shape and constrain every other decision down the line. Similar "up front" decisions around your expected customer experience influence your ability to update or change your product after it's already in the customer's hands. These are bookends if you will; decisions that frame the entire development process.

At the beginning: avoiding bad decisions

The hardest product decisions to work around are poor hardware choices. If you design a chip into your product that lacks in processing power, memory, flash disk, configurability, or sufficient I/O capabilities, you're consigning your software team to extremely costly engineering to work around these limitations. Ill-chosen chips can lead to fewer features, slower performance, and intermittent failures, creating a significantly less satisfactory customer experience. You'll also certainly create a less solid code base due to the kludges needed to sleight-of-hand your way out of jams. The worst scenario is that these issues are intractable enough that the hardware needs to be designed out, resulting in months of lost time and mounting costs.

The fundamental takeaway is you must pick hardware that meets your requirements both now and in the future.

It's easy to mistake something as "technically possible" by the processor specs, but that isn't achievable when considering a holistic view of the product, constrained developer deadlines, or ongoing maintenance considerations.

When the BOM becomes a bomb

Most bad hardware decisions are the result of pursuing hardware cost savings at the project onset without the full realization of that choice's potential impact to software development timelines and resource increases. Purchasing and bill-of-materials (BOM) concerns can steer you into short-sighted savings unless you ensure that software experts are sitting at the table and can validate product choices.

From experience: Select your hardware to grow

Picking a "right-sized" CPU and board for your project can be a limiting long-term choice. We worked with one customer who made agricultural machine controls, and their original hardware choice worked acceptably for the first product. But over time, they added many new features such as more complicated applications, 3D visualizations, and camera inputs. After every new feature, they had to spend huge amounts of time hunting for small performance gains so they could squeeze in the new functionality or add new code to the flash drive. The amount of time their engineering team spent on these issues reduced their ability to innovate and get products to market. And even with all the performance gains squeezed out of the system, they still had to live with many unfortunate limitations such as a slow camera frame rate.

Supporting specialized hardware

Things like physical buttons, industry-specific hardware, or specialized sensors all need to be connected to the system whether directly through GPIO, A/D, or UART inputs, or a bus like I2C, USB, or CAN. Make sure you have the required hardware on your board and, if it needs a gateway, you have enough spare ports to manage everything you need.

If you're reading and writing to hardware through a GPIO, this requires a dedicated pin – a very limited resource. The number of GPIOs are often changeable through configuration registers

(consult your CPU's reference manual) but only at the cost of other possible features. Figure out how many GPIOs you need and see if that configuration is possible with your board while accommodating your other hardware requirements.

Are there drivers that the board vendor supplies for talking to your devices, or do you need to write them yourself? If the drivers aren't available out of the box, there may be source that can be easily adapted. Even so, writing kernel drivers is a specific skill that demands more rigor with different APIs and tools. You'll save lots of time and heartache if you can find professionals who specialize in this sort of thing.

From experience: Extra peripherals can come in handy

You might be looking at boards that have extra hardware you don't expect to use, like a Bluetooth chip or USB port. In our experience, many customers have been able to use those features for future customer and development features, even when they weren't part of the original design. You might not need that USB port for the final product, but it might come in very handy when your developers use it for an extra network or WiFi dongle or memory stick during development. If your board has Bluetooth support but your product doesn't, having the capability as an option lets you add a Bluetooth antenna later when your customer wants to pair their phone to your device.

At the end: shipping the product

You're not done when you're done coding; you still have to have a systematic, reliable way to get your software into customer's hands. Putting software loads onto boards before you've shipped them isn't too difficult, since you can rely on specialized tools and the same software environments you've been using throughout development. Updating products after they're in the customer's hands is fundamentally more difficult – and it's expected. Because there are many more considerations for software updates, we'll focus on this area for the rest of this section.

Deploying images

How are you deploying development images? Whether user-initiated or automatic, most products today allow in-field product

updates. User-initiated updates require less work for the developer (and more for the user) than automatic ones. But they may be a first step – your website can provide firmware download images, and you can supply some on-board scripts to detect and install those images from an on-device upload, USB stick, or desktop PC.

Be careful in any type of update procedure that corrupt images or process interruptions don't brick the unit. Using dual A/B firmware loads can help ensure you always have at least one part of the unit booting properly. There are also commercial and open-source solutions for remotely updating software: in other words, over-the-air (OTA) updates. These solutions are quite a bit more robust in handling failures and interruptions.

Should you auto-update?

You'll want to carefully consider if auto-updating is right for your device because while it ensures your customer always has the latest features and security patches, enforcing periodic downtime to handle the updates can have a big impact on how your customer is able to use the device. In all cases, you want it a controllable feature so that the customer doesn't get stuck in a download or updating situation when they least desire it. Any embedded device that is expected to be continually in use or that has stringent security requirements should either only use manual updates or at the very least have the auto-updating feature default be off.

Updating without breaking

Regardless of how you get new software to your device, your software team must be very careful that user data isn't destroyed (from such things as file format changes) and that configuration files are maintained. Develop a clear way to version data files and automatically migrate them forward (or auto-migrate files bidirectionally if you decide to allow older firmware downloads). For your configuration files, use a simple text format like JSON

where you can supply default values to accommodate missing data fields – and perhaps even allow the user to manually fix configuration files if they get broken or if they need to coordinate internal deployments of your products.

Preparing your image files

Every release needs a unique version number – but you're already generating that as part of your build, right? You need to compress your release images to save space in both downloading and in temporary space on the device. You also need to wrap all release files with a CRC – or perhaps a digital signature – so they can be verified and not easily tampered with. And you want to maintain an in-house library of all releases so that you've got easy access to the binary files when you need them for testing. Of course, you also need every customer-facing release tagged within your source control system for easy replication of any historical code snapshot when necessary.

And while most of this discussion has been around firmware images, if you're using Docker or another container-like system, you still need the same basics: how does your customer get the files, how do they get verified, and how are they reliably put into service?

Quality assurance

Of course, your developers are testing bits and pieces as they go. But who is testing the final product? Is that testing going to be in-house or outsourced? And – although nobody ever wants to admit it – how will you handle bug reporting when the customers have been left to test the hardest parts? Can you recover your customer from deep failures with a factory reset and/or default firmware? And how can you guarantee that the factory reset will be complete – including resetting any non-obvious but persistent hardware or file system states? Knowing the answers for how your testing, support, and customer interaction should work together is important before you start shipping.

Similarly, you want to think through your software update process. Are you going to be fixing bugs and security issues on a regular cadence, when customers prompt you for fixes, or when discussed as part of your engineering roundtable meetings? Are there some customers who need specialized "on-demand" branches for critical bug fixes, because they're your biggest concerns or they require it contractually via their support agreement?

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 first 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 second, third, 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