

Designing Your First Embedded Linux Device 4

The Development Environment

Nathan Collins | Senior Software Engineer, KDAB

 KDAB



Before creating your first embedded Linux device, you need to determine how you want to build your software. After all, the tools and development processes for embedded Linux may be quite different from what you're used to with smaller microcontroller-based or bare-metal systems. If you want a well-designed, easy to maintain, and cost-effective solution, take the time to research your options and keep in mind the following key considerations.

Programming language

If you've been working with a PIC or microcontroller, you've probably been using C or even a custom C-like language. You could still do C for your Embedded Linux application, but there are many more options available – many with more modern features that make coding more concise, concurrent, and fail-safe. Let's look at three of the most interesting choices for embedded development.

- **C++.** This is the clear successor to C although it scares people off with its infamous complexity. Yet with its unparalleled flexibility, excellence in creating optimized code, and ability to access low-level internals, C++ is the go-to choice for most embedded engineers. It's also the choice for many library vendors since APIs for most third-party content are easy to access. The C++ standards body has been working hard to make the language easier and more foolproof, with newer, simpler, and better programming constructs and idioms in C++11, C++14, C++17, and C++20. If you're considering a "traditional" route and using C++, you owe it to yourself to brush up on these newer C++ standards. Modern C++ has things like automatic strong typing, template metaprogramming, and lambda functions, making the bulk of coding much more pleasant, and leaving gory details still accessible when you need them.

- **Rust.** Relatively new to the embedded scene, Rust is another consideration to create your embedded software stack. It has some fantastic bug-avoidance features built-in to the language, such as guaranteeing safe memory accesses and safe concurrent thread operations. It also creates fast code, can access low-level hardware, and is compatible with C APIs, just like C++. The Rust ecosystem isn't yet as large as C++, but with these attractive features, we are seeing more interest and uptake.
- **Dart.** The newest language competing for your embedded software source is Dart, which is a programming language created by Google. Similar to other modern languages like Kotlin or Swift, it is designed to maximize productivity while minimizing programmer error. Unlike those options, Dart is cross-platform and includes a cross-platform UI option, Flutter. We expect Dart/Flutter use within embedded to keep growing.

Libraries and dependencies

Most software projects aren't completely in-house because it doesn't make sense when there are so many tested open-source options that others have already built to do the hard stuff like regular expressions, context-free parsing, fast-Fourier transforms, digital filters, matrix math, machine learning, encryption, authentication, and many more.

So, the biggest question is – are any of the tools you need already supported on your chosen platform? They can be part of a distribution already or perhaps enabled by a configuration line or two in your yocto build. Or, you can get any necessary components on your system via a package manager (like apt-get, rpm, or yum) or a universal distribution manager (like flatpak or snap). For a first time Linux build, you probably want to avoid getting everything yourself. Afterall, it's the modern era; let tools get the source zips and dependencies that you need.

Licensing

When building your embedded Linux software stack, a smart engineering team will leverage as much software as it can from multiple sources. Much of this will be open source and, while that's great, staying on top of the licensing requirements is also something that's easily overlooked. There are many common licensing policies with differing requirements and/or restrictions, such as GPL 2, GPL 3, LGPL, BSD, MIT, and Apache. Your licensing terms may require you to publicly list contributions, make source code available, or submit back fixes – among other obligations.

It's important to have someone (or something) track all the chunks of software that are going into your build. Dedicate someone on the engineering and legal sides for this to make sure that your company is adhering to licensing terms properly. You can automate some of this too: companies like Synopsys (with Black Duck) and BlackBerry (with BlackBerry Jarvis) offer software solutions that scan either your code base or your executables to provide a comprehensive view of your third party components in use and their requisite licensing responsibilities.

Yocto also produces a license manifest for you where you can allow or deny certain types of licensing models as your corporate needs dictate. Given the tedium of this task, these types of tools can often examine your code more thoroughly and consistently than a person would.

Multi-platform

The software you're building may need to run on more than one target. A few common examples:

- **Emulation/Virtual Machine.** If you do much of your testing and debugging on a PC, you'll need an "emulator" build that lets you run your application in a virtual machine in a Linux desktop or container. This will output an executable rather than a build

image and will need to provide custom low-level software components that can emulate or fake dedicated hardware.

- **Multi-board.** Your product may have different options (such as screen size and/or capabilities) that drive it onto different boards. Each new board variant needs a dedicated build, which can greatly expand the amount of time and space your build consumes.
- **Mobile.** Some of your software may run on a companion mobile app, especially if there is a remote configuration, reporting, or user interface component. Since “bring your own screen” is a simple way for manufacturers to reduce cost, your product may need a downloadable app that accompanies your product and that shares some of the same libraries or product definitions. These companion mobile apps may require you to build some iOS and Android components, if not the full application, at the same time you’re doing the embedded builds to ensure they’re paired together properly.

If you’re building on multiple platforms, decide if you need to cross-compile every platform on every commit. While this ensures you’ll always have up-to-date software for every variant, it can lengthen the full build cycle significantly. You may decide to reserve certain build platforms for a daily (or longer) build cadence.

Building and testing

You need to set up a build system, a dedicated build computer, build scripts, and notification systems. Make sure that one person is definitively responsible for the build system and build infrastructure.

Since it’s the easiest way to quickly spin up, many systems start out with a traditional build (especially with a small development team). But moving to a continuous integration (CI) system has its benefits. CI makes demands on scaffolding that can seem like a

lot of extra effort such as an automated build, full unit testing, and automated test scripts. However, these methodologies significantly help move the software forward with fewer bugs, more reliable releases, and more confident and speedy maintenance changes. If you're developing with sufficient rigor anyway, most of what you need to implement CI will already be in place; creating unit tests and automated test scripts should be part of your development timeline and having an automated build procedure is mandatory to ensure reliability.

Besides just automated or unit tests, you should also do integration testing and user interface testing. Although a bit harder, these too can be predominately automated. And don't fall into the trap of assuming everything that runs properly in an emulated desktop environment will work as smoothly on the board. Your test scaffolding should include hardware target download and test cycles too.


Debugging

Finding software bugs can be a challenge during development and is much trickier when problems are only encountered in the field. Since every tool has situations where it works well and others where it doesn't work at all, you want to get as many debugging tools as possible to help.

- **Hardware.** While a company usually has limited access to hardware tools like digital oscilloscopes, you can find decent, simple scopes on eBay for a reasonable amount of money. There is nothing as concrete as a scope for confirming some hypotheses; in fact, they can be invaluable. Even simply monitoring a single pin output with a scope can verify if code gets to a certain critical point without introducing the timing delay of a log. It can also verify if your interrupt is getting triggered multiple times and can confirm precise timing and correlation between two events. A knowledgeable engineer with a scope can do much more.

- **Debugger.** This is often a GUI-powered debugger that's part of an IDE, but it can also be a standalone application or even a text-only interface. Whether you're debugging on-target or executables in an emulated/virtual machine environment, a debugger allows you to watch code execute, set breakpoints, monitor or change variables, and see thread activity – all extremely helpful for figuring out issues. There are a number of situations that are problematic for debugging, however, regardless of whether it's on-device or on your desktop. Anything with timing dependencies, from hardware timing responses to microservice timeouts, are nearly impossible to debug in this environment. You may be able to make a special build that disables all timeouts and sets hardware timers to their maximum – but this obviously changes the software and can make heisenbugs go away. Similarly, optimized code rarely has a one-to-one correspondence to source lines of code. This means that to understand code execution paths, you often need to debug using a non-optimized build, and the changes it makes to the resulting executable can make subtle problems disappear. (This is why you need a few different tools in your tool chest.)
- **Static analysis.** Static analysis isn't good for finding an isolated bug but rather for finding many undiscovered bugs. There are many of these tools available – from commercial to open-source offerings – and they can make a tremendous improvement to your code quality. The biggest challenge is often adding them to an existing code base, because they can flag a massive number of issues that need to be investigated while often only a tiny fraction of these turn out to be legitimate issues. If you're starting out, we recommend adding a static analysis tool from the very beginning so that your software is continually being checked as the code base accumulates in size. This maximizes your benefit and prevents the pain from being experienced all at once.

- **Run-time checkers.** A run-time checker is used for types of hard-to-detect behavior that can be caught by instrumenting the code. One prominent example is to track memory allocations, ensuring that they're all correctly allocated, freed, and not improperly overwritten. It can also be used for things like run-time array index checking or system API parameter checks. All of these applications obviously carry a run-time penalty, but because they're so useful to find certain classes of bug, they should be a definite consideration for your toolkit.
- **Application profilers.** These tools aren't usually used to find bugs in the literal sense. However, a performance failure is a defect that requires fixing before release, and an application profiler can help you understand where and how code needs to be changed to extract more speed out of critical bits of code.
- **System profilers.** This profiler lets you see interactivity between multiple applications running in the system. A system profiler can also show interactions with hardware (such as disk writes or interrupts) that initiate certain software behaviors. While system profiler traces can be complex to interpret, they are the tool of choice for full-stack gurus since they provide a holistic view of system execution.
- **Logging.** The simple "printf" has long been the bare minimum of debugging assistance, and it remains a well-used tool for a few reasons. Logging can be added exactly where needed to confirm or investigate code flow and/or variable content. Logging can be left in production products allowing in-field failures to be partially diagnosed. And because it is dead simple to implement in its easiest form, logging is always the first debugging assist. However, logging often ends up gathering many more sophisticated requirements, like differing log levels, module-controllable enablement, log file rotation, caching and file flushing, and remote retrieval. Excessive logging can introduce issues too, from excessive disk storage or wear, slowing down parallel file system accesses, introducing



significant timing delays, and other unexpected side-effects. While it's tempting (and perhaps fun) to create your own logging library, look around for existing ones that meet your needs first. (Of course, check these for licensing models that are compatible with your legal department.)

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 fourth 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 third 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

© 2020 the KDAB Group. KDAB is a registered trademark of the KDAB Group. All other trademarks belong to their respective owners.

