KDAB's Software Development Best Practices

Embedded Development

Nathan Collins | Senior Software Engineer, KDAB

KDAB

The software engineering community has learned hard lessons over time about the best way to build software. Following these tried-and-true practices provides a huge number of benefits: more resilient software, faster release schedules, higher quality products, more effective teamwork, and happier developers.

Unlike a lot of modern development, embedded development is close to the hardware. It runs on specialized boards, uses custom testing harnesses, requires expensive debugging equipment, and has its own characteristic workflows. That hardware dependency makes embedded development unique. Here are a few of the best practices that we try to use in our embedded development projects.

1. Hardware

1.1. Board selection

We've talked about this in our whitepaper, Designing Your First Embedded Linux Device, but hardware selection at the beginning of an embedded project is critical. The choice very often uses cost as one of the primary selection factors. Which of course makes sense; every dollar you spend on product manufacturing is one more dollar you must charge your customers, making you less competitive, or one less dollar you can realize in profit.

Nothing comes for free though – that cheap hardware does come with a cost. Your product's software can only use the resources that are available. Just be aware that when you minimize hardware bill-of-material (BOM) costs, you are likely making for more expensive software, longer development time, or a poorer user experience. We think you should **consider more than price when selecting your hardware**. Specifically, weigh your hardware selection against potential development timelines, engineering effort, and eventual customer impacts. You can do this with specific feature prototyping to gauge whether your product's feature set will fit the desired hardware platform. It's a good idea to give yourself a bit of head room – more CPU, RAM, storage, and capability than you might minimally need – so you can implement new features and product refreshes without a hardware replacement.

BUDGET HARDWARE EQUALS BUDGET UX

No matter what device people use today, it's a given that they expect a smartphone-like experience with a smooth and sophisticated UI. Most embedded manufacturers also want that polished look for their own products. This includes one of our customers who went with an inexpensive, relatively lightweight processor on a third-party recommendation. However, smart phones aren't designed with a lightweight processor – they're loaded with a fast, capable ARM64 multicore processor, a high-powered GPU, and plenty of RAM.

We were brought in to help because their software team was struggling for months to make that underpowered chip deliver the smooth animations and subtle gradients their UI team had designed. After some analysis and targeted tests, we determined that there was no amount of optimization that could deliver smartphone-like features due to constrains on processing power and a lack of hardware accelerated graphics. Instead, we recommended a drastically simplified UI that eliminated all the bells and whistles. While this client ended up with a functioning product in the end, they had to settle for a poorer UX than they had originally envisioned.

1.2. Timing and availability

Embedded software ultimately lives on embedded hardware. That means access to the real production hardware will always be a gating factor. Tools like emulators and virtual machines (VMs) can mimic the real thing, but they aren't identical to your hardware running in a real physical box; the peripherals, performance, characteristics, and environment will always be subtly different between these tools and the real thing. This can hide performance and integration issues until late in the development cycle. Plus, VMs require a different set of drivers and configurations than the target hardware and are likely to have limitations on what can be tested since some things aren't easily emulated.

All that leads most embedded companies to get hardware for the developers as soon as possible. That's a necessary first step, but we also recommend that you **acquire adequate hardware resources for testing purposes**. Early testing on hardware is critical to identifying performance or integration issues, which desktop simulation or cross-platform builds aren't good at uncovering. Your test team needs access to that precious and limited hardware resource to be effective, just like a properly functioning CI/CD test environment does. Don't create two classes of engineers – with and without hardware – as that will naturally inject delays and scheduling conflicts.

2. Tools and tooling

2.1. Automation

We've talked about the importance of automation in our Software Development Best Practices: General Development whitepaper; in fact, it's the first recommendation we make in it. Embedded development has some singular attributes that make this even more important. It uses special tools, applications, and processes for accessing the hardware. These each contribute to longer times for developers being productive and require more expertise.

You can eliminate a lot of the the hardware-incurred developer penalties if you **automate all hardware-specific tasks**. We're talking about things like putting the application on the board, starting the debugger, resetting the application in a fresh state, gathering log and crash data, loading up particular configurations, and restoring the OS from a known-good image. Anything that you need to onboard a new engineer should be automated, as well as all daily development tasks and anything required by the test workflow.

3. Development

3.1. Prototyping

We touched lightly on this in section 1.1, but prototypes are essential for embedded. Because switching software and hardware strategies late in the project are exponentially more costly, you need the quick insights into specific concerns that prototypes give you to iron out trouble spots early in the project. Your prototypes don't have to mimic the final product or large, complex endeavours. Instead, **develop targeted prototypes that run on the embedded hardware** to confirm and fine-tune questions about your hardware selection, software performance, integration strategy, and UI design. Some examples of reasons we might prototype:

- Checking overall performance characteristics of a particular combination of hardware and software framework to ensure they fit design expectations

- Benchmarking critical hardware functionality to see how well execution speed on a system/board compares against chipset data sheets

- Validate user workflow and user experience questions with a "dummy" UI to point out areas of confusion or invalid user assumptions

- Estimate memory high-water marks to get a guess about memory sizing or an early warning on memory consumption issues

Your prototypes give you confidence that you can identify issues before they become show-stoppers. Prototype code doesn't have to be throw-away either – if you're using it to validate your hardware or software, you can bring it back out to retest if you change the underlying components.

3.2. Integration

A big source of heartburn can be the integration step – you're throwing away the scaffolding and weaving together software that's been developed separately. Lots of pieces of your own software are often built with separate teams: software stack, user interface, main application, backend, services, etc. It's always tempting to put off the pain. That's true too with third parties whose software update cadence might not match yours. However, we recommend that you **integrate early and often**. That gives you time to identify and fix issues that cannot be found in individual unit tests and might otherwise not be found until pre-release.

3.3. Optimization

Most programmers have heard "premature optimization is the root of all evil," a saying that's been around since Donald Knuth uttered it in the 1960's. So why are we still doing it? It's common that we find developers optimizing algorithms without clearly identifying specific performance problems first. Nowhere is this more important than on embedded devices where many different hardware and software subsystems can contribute to performance issues.

We're not saying don't optimize, but rather **don't optimize until you fully understand the problem**. Use profiling tools like perf and hotspot, system-level profilers, and memory profilers to be certain you know what performance issues you're facing. Then you can tackle the problem with a clear strategy and develop benchmarks to detect and prevent future regression.

BYTE-WISE AND MEGABYTE-FOOLISH

We worked with a customer who ran into memory constraints and had highly optimized their application's search function. It required 200 entries maximum, and they were able to trim teach entry down to 1Kb for a total maximum of 200Kb. However, they were still running into memory issues. We profiled the application's memory and noticed that they had accidentally used a wasteful file format for a background image that was needlessly taking up 12Mb. Fixing the background image with a simple image conversion could have given them an easy win and saved their team a huge amount of work.

4. Architecture

4.1. OS selection

Embedded projects come in all shapes and sizes. But if you know how to whip up a Yocto project build image, you've got yourself a good hammer for every new project that now looks like a nail. But don't fall into the trap of using that same old hammer; it's important to **pick an OS that's appropriate to your project**. What does that mean? We're not going to get into the technical weeds on the OS selection but here are some things to consider:

- **Should you use Yocto?** Yocto is amazing and we use it regularly. But it outputs a custom Linux distribution. That's a lot of components and configuration that you might not need that just add complexity to your project. You might be able to get away with a simpler install and switch to Yocto if project flexibility becomes a priority.

- **Should you use buildroot?** One step down from Yocto is buildroot. If you don't need ultimate configurability of everything and need a simpler tool to manage your OS, then buildroot is a better choice.

- **Should you customize your OS?** If you're using off-the-shelf boards, maybe the OS install that's already on them will work if you don't need anything special. If you can add a driver you need and those couple of missing packages, maybe that's enough. Sometimes you can get exactly what you need by packaging your application in containers or in a package suitable for your Linux distribution.

- **Should you use Linux?** Maybe you don't even need Linux. If you've got special certification needs, require hard real-time, or need to run on severely limited hardware, you'll probably need another choice of operating system, tool chain, and vendors.

5. Testing and debugging

5.1. Test cadence

We know that desktop processors and CI/CD infrastructure are going to be so much faster than the product hardware that you'll get test results back in a fraction of the time. It's the nature of the beast. But you need to **regularly run your test and benchmark suite on your embedded hardware**, at least nightly. Otherwise, you might not notice that the team has introduced regressions into software during development until much later.

5.2. Full stack

Understanding the full software stack – hardware, OS, drivers, services, libraries, application, and user interface – is always helpful. But in embedded there is nobody else to pass the buck; your team is responsible for debugging and fixing any issues. That's why we know that to be successful, you must develop **the expertise to debug and troubleshoot**

throughout the full stack. Be comfortable working within the stack and understand which tools can target which layers, like strace, heaptrack, or lttng.

5.3. Hardware Abstraction

Almost by definition, an embedded system has hardware – GPIOs, A/Ds, USB-connected modules, SPI devices, you name it. However, because you won't always be running your application on an embedded board, you should **have a hardware abstraction layer (HAL) for your peripherals with a software mock-up behind it**. In some cases, you might need to design an API that can hide the hardware first. (It doesn't need to be complex or handle all cases; something simple is sufficient.) In other cases, you may be able to leverage an already existing clean API that abstracts away the hardware. In either case, you'll need to create a set of suitable software stubs that mimic the hardware interface.

BLACK IS NOT THE NEW BLACK

One customer of ours had prototype hardware that ran perfectly. However, once they received their first pre-production run of 150 devices, they tried to fire them up. The final production displays were supposed to be brighter, but wouldn't turn on – they remained black. They called us in a panic – once we had done some digging, we discovered that the Linux GPIO driver had a setting that wasn't compatible with one of the display pins firing up the display. A 100ns timing change was all that was needed to get the displays working properly.

All you're trying to do is ensure that you preserve the ability to build and run your application in a desktop or cloud environment. Of course, you don't need to replicate the functionality of the hardware – that's overkill. Just add enough logic so the calling functions aren't surprised with unexpected sequences of states or bad return codes.

You'll probably want to skip testing any of these stubbed-out "hardware" routines in desktop runs. However, be sure you test the real hardware APIs in nightly CI runs so you'll still get enough warning if something breaks due to code changes during the day.

What is KDAB's Software Development Best Practice series?

This series of whitepapers captures some of the hard-won experience that our senior engineering staff has developed over many years and projects. Offered up as a grab bag of techniques and approaches, we believe that these tips have helped us improve the overall development experience and quality of the resulting software. We hope they can offer the same benefits to you.

View all three parts of this whitepaper series online at: ***www.kdab.com/publications/bestpractices/***

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

KDAB

CERTIFIED ISO 9001:2015 COMPANY