Configuring the Application Development Environment

Pick the Right Tool for the Job

Using Linux grants a great degree of freedom when you're selecting an approach to use when implementing a project; the same tools available on a desktop are available for an embedded project. The great variety of tools is very liberating for engineers who could only code in C in years past—now you can use C++, Perl, C#, shell scripts, Python, or Java to build an application. Picking the right tool for the job is an important decision that involves knowing the nature of the application and the hardware constraints.

Not only does Linux give you great freedom in selecting development tools, but the environment also doesn’t limit the selection to one. Using a mix of C and Java is a perfectly good idea if the device will be communicating with an enterprise server that’s running Java. It’s also entirely practical to use C++ as the tool for the UI development while using a mix of shell scripts and Perl to process data. When you’re selecting what to use for application development, the key isn’t to focus on picking one right tool; the important part is to identify what the application is doing and then choose the correct development tools for different parts of the job.

This chapter starts by discussing the importance of understanding requirements: many projects get off on the wrong foot and never recover because the goals are never well understood by all involved. That results in the wrong choices being made. Embedded development can have some interesting twists and include requirements not found in other projects. Knowing about those twists before you start helps you identify risky parts of the project sooner. To paraphrase a random U.S. politician, “There are unknowns and unknown unknowns.” This chapter makes both less mysterious.

Know Your Application

Knowing your application means understanding the requirements of the project. The requirements, in the software world, specify what the application must do. The process behind requirements is a book in itself;1 however, the Zen of requirements boils down to the following:

1. Requirements are written.
2. Changes are tracked.
3. Project participants understand the requirements.
4. The requirements explain what and not how.

---

1 I recommend reading the excellent Managing the Software Process by Watts Humphrey (Addison-Wesley Professional, 1989).
When you're an engineer working on the project, items 3 and 4 are very important. Many times, the product manager responsible for the project doesn't supply sufficient detail: for example, “The device must boot quickly” as opposed to “The device must be ready to process data in 10 seconds or less.” Or you may get conflicting items like “The device must store 10,000 audio files” and “The device must have no more than 64MB of flash.” The last requirement brings up the most obvious difference in embedded development: the additional constraints introduced by the hardware itself.

Hardware Constraints
When you’re starting on the requirements for an embedded project, divide the list into hardware versus software constraints. Hardware requirements represent the most constraining items because they reflect equipment already in the field that can’t be replaced, hardware that has been designed, or market expectations where users won’t accept any other solution.

Some hardware constraints are easy to spot (touch screen, wired network connectivity, CANbus support), whereas others are more subtle. For example, the following place hardware constraints on the project that may not be intuitively obvious (some are real-life examples and others are made-up—see if you can tell):

- **The device will operate for 6 hours once disconnected from commercial power:** This was an actual requirement. It was later refined to include a statement about what sort of battery would be attached to the device. This requirement could be solved by attaching a car battery to the device, which was not the intent. After clarification regarding the type of backup power that could be accommodated and the notion of “operate,” it was clear that the device needed to be a low-power chip with power management and not that much RAM memory, because that’s a drain on power. This influenced the language selected (C, due to the low amount of memory) and the kernel (power management was key), which had a cascading effect throughout the project.

- **The unit will emit no more than 15 BTUs per hour:** Some equipment has a power budget, and the corollary to that is a heat budget. This is another requirement that affects the amount of RAM, CPU MHz, and type of power management that must be available. These sorts of requirements appear in all sorts of projects, and it’s no accident that cell phones don’t get too warm in your hand. This was a real customer requirement.

- **The product will produce 0 decibels of sound at 3 feet:** This means a couple of things: no fans for cooling, and no hard drive. The device must use flash memory. Home entertainment systems have requirements like this because nobody wants to hear a fan whirling during the favorite parts of their movie. Automotive devices fall into this category too, because background cabin noise is a competitive factor.

- **The system will make a routing decision in no more than 2 microseconds:** This is a sneaky requirement, because this is really a realtime system. **Realtime** means that if a deadline isn’t met, the output of the system is incorrect. Realtime systems need to have software that works deterministically; but in order for the software to function as expected, the underlying hardware must also work in a deterministic manner. In this example, at a minimum, the system clock must have a minimum amount of drift and the hardware interfacing with the peripherals must also work accurately. **Drift** occurs when the system clock is running ahead or behind the real-world clock. This is not a traditional wall clock but the hardware clock used to drive the processor. When this clocks ticks at an irregular rate (as some do), it drifts out of time compared to a more accurate clock.