A System is not a Product

Stopping to smell the code before wasting time reentering configuration data

Every once in a while, I come across a piece of good code and like to take a moment to recognize this fact, if only to keep my blood pressure low before my yearly medical checkup.

The first such piece of code to catch my eye was `clocksource.h` in Linux. Linux interfaces with hardware clocks, such as the crystal on a motherboard, through a set of structures that are put together like a set of Russian dolls.

At the very center is the `cyclecounter`, a very simple abstraction that returns the current counter from an underlying piece of hardware. A `cyclecounter` knows nothing about the current time, time zone, or anything else; it knows only what the register in a piece of hardware is when asked about it. The `cyclecounter` has two pieces of state that help translate cycles into nanoseconds but nothing else. The next doll out is the `timecounter`. A `timecounter` contains a `cyclecounter` and raises the level of abstraction to the level of monotonically increasing time, measured in nanoseconds. On top of these structures are others that eventually give the system enough abstraction to know what the time of day is, etc.

So, what is so great about this code? Well, two things: first, it is well structured, in that it is built from small components that can cooperate without giving each other a reach-around or a layering violation; second, it is written and documented in a style that is clear and clean enough that I was able on first reading to understand how it worked.

The comments and structure for the `cyclecounter` give you a flavor of what makes me so happy to read this code:

```
/**
 * struct cyclecounter - hardware abstraction for a free running counter
 *   Provides completely state-free accessors to the underlying hardware.
 *   Depending on which hardware it reads, the cycle counter may wrap
 *   around quickly. Locking rules (if necessary) have to be defined
 *   by the implementor and user of specific instances of this API.
 *
 * @read:      returns the current cycle value
 * @mask:      bitmask for two's complement
 *             subtraction of non 64 bit counters,
 *             see CLOCKSOURCE_MASK() helper macro
 * @mult:      cycle to nanosecond multiplier
 * @shift:     cycle to nanosecond divisor (power of two)
 */
struct cyclecounter {
    cycle_t (*read)(const struct cyclecounter *cc);
    cycle_t mask;
    u32 mult;
    u32 shift;
};
```

I think you can see why I like this code, but just in case you can't, let me be more specific. The code is well laid out, nicely indented, and has variables that are short, yet readable. There are no Bouncy Caps or very_long_names_that_read_like_sentences. The comment is long enough to describe not just what the structure is, but how it is used, and it even mentions what has to be done if multiple threads need to access one of these structures simultaneously. If only all code were this well documented! You can read more of this code online at

<http://lxr.free-electrons.com/source/include/linux/clocksource.h>.

My other example of good code requires more explanation, so I'm going to reserve it for a future column. After all, I don't want to waste the calming effect all in the same issue.

Dear KV,

After spending the last year buying and deploying a set of monitoring systems in my company's production network, we hit our first bug in the manufacturer's system software. After we reported the bug, the manufacturer asked us to upgrade the systems to the latest release. It turns out that the upgrade process requires us to reset the devices to their factory defaults, losing all our configuration information on each system and requiring a person to reenter all of the configuration data after the upgrade.

At first, we thought this might be something required only for this particular upgrade, which crosses a major revision, but it turns out that we will have to reenter our configuration data each and every time we upgrade the software on these systems. I guess we should have asked this question before we bought the systems, but it just never occurred to us that anyone would sell a box purporting to act as an appliance that could not be easily upgraded in the field. One of the other guys in my group suggested we just return the boxes and ask for our money back, but, depressingly enough, these are the best systems we've found for network monitoring.

Down on Upgrades

Dear Down,

It seems that what you thought was a product—that is, a set of components thoughtfully put together by people who care about their customers—was actually just a collection of parts that under normal circumstances worked well enough to get by. Unfortunately the number of companies that think about what's going to happen after version 1.0 of their systems is quite small.

I've been fortunate—or perhaps I should say that the sales critters I've dealt with in the past 10 or so years have been fortunate—not to come across too many systems that act in the way you describe. The idea that the manufacturer would require a user or systems administrator to reenter already-saved data after an upgrade is stupid, ludicrous, and a bunch of other words that my editors just aren't going to allow in this publication.

Even in the worst-designed products—and I've used enough of those—there is usually some bit of perl that takes the old configuration data and turns it into something that's mostly valid for the latest revision.

As a matter of fact, many years ago I worked on a networking switch project, and the first thing the systems team (which was responsible for getting a reasonable operating system and applications onto the box) did was come up with a way to field-upgrade the system. Anyone who has ever configured a switch or router knows that you don't just toss out the configuration on an upgrade.

The sad part is that doing this correctly just isn't that difficult. Most embedded systems now use stripped-down Unix systems such as Linux or the BSDs, all of which store their configurations in well-known files. Granted, this isn't the nicest way to store configuration data, because it tends to be a bit scattered, but it's not that hard to write a script that can handle differences between the versions and reconcile them. On FreeBSD there is `etcupdate`, and Linux has `etc-update` and `dispatch-conf`. In a properly designed system the configuration would likely be stored in a simple database or an XML file, both of which are field-upgradable with fairly simple scripts.

These sorts of issues are what differentiate an *appliance* that can be deployed and maintained with little human interference from a *system* that has to be diddled constantly to keep it happy. It is a sad fact that many programmers and engineers do not think much of appliances and are more likely to think that their users should “man up” and spend their time making up for the original designer's lack of thought.

I still remember one of the first all-digital stereo systems I ever saw. It was designed around a Sun workstation and cost on the order of \$15,000. It was obvious from the moment you looked at the controls that little or no thought had been put into what people really wanted out of a sound system. What most people want out of a stereo is good-quality sound with a minimum of button pushing to get what they want. What the system I saw presented was a lot of button pushing for about the same quality of sound that I could get out of an amplifier and a CD player. If the user interface was horrific, it was nothing compared with the system performance. The box crashed three times before I finally walked out of the store. The one thing I got from that experience was an understanding that systems and products are not the same thing.

A *system* is a collection of components put together to do a job. A *product* is a system that has been designed and built to make the work of carrying out the job smooth and natural to the user. I can certainly cobble together software to rip, store, and play my CDs on a computer; that is a system, whereas an iPhone is a product. When I upgrade my phone, I don't reenter my data. If I did, the product would have died at the first revision. More likely, Steve Jobs would have taken someone's head off if he had been told that the upgrade path required reentering data. Given how complex modern appliances are—and let's face it, your TV probably has an Ethernet jack—it's clear that people have thought about and solved this problem.

The real issue is with the people who design these devices. Somehow the fact that an appliance is going to be hanging out with a bunch of computers—for example, in a colocation—makes it acceptable for the implementers to make their box look more like an open source desktop, which will be diddled by an experienced IT person or the users themselves. It's a practice that really needs to stop, if only because I don't want everyone to wind up bald, like me. My hair didn't fall out, I pulled it out!

KV

KODE VICIOUS, known to mere mortals as George V. Neville-Neil, works on networking and operating system code for fun and profit. He also teaches courses on various subjects related to programming. His areas of interest are code spelunking, operating systems, and rewriting your bad code (OK, maybe not that last one). He earned his bachelor's degree in computer science at Northeastern University in Boston, Massachusetts, and is a member of ACM, the Usenix Association, and IEEE. He is an avid bicyclist and traveler who currently lives in New York City.

© 2012 ACM 1542-7730/11/0400 \$10.00