BEAUTIFUL CODE EXISTS, IF YOU KNOW WHERE TO LOOK

A koder with attitude, KV answers your questions. Miss Manners he ain't.

Dear KV,
I've been reading your rants in *Queue* for a while now and I can't help asking, is there any code you *do* like? You always seem so negative; I really wonder if you actually believe the world of programming is such an ugly place or if there is, somewhere, some happy place that you go to but never tell your readers about.

A Happy Programmer

FIGURE 1

```
/*
 * struct pmc_mdep
 *
 * Machine dependent bits needed per CPU type.
 */

struct pmc_mdep {
        uint32_t pmd_cputype;     /* from enum pmc_cputype */
        uint32_t pmd_npmc;        /* max PMCs per CPU */
        uint32_t pmd_nclass;      /* # PMC classes supported */
        struct pmc_classinfo pmd_classes[PMC_CLASS_MAX];
        int pmd_nclasspmcs[PMC_CLASS_MAX];

        /*
         * Methods
         */

        int (*pmd_init)(int _cpu);   /* machine dependent initialization */
        int (*pmd_cleanup)(int _cpu); /* machine dependent cleanup */
```

Dear Mr. or Ms. Happy
While I will try not to take exception to your calling my writings "rants," I have to say that I am surprised by your question. KV is a happy, cheerful, outgoing kind of guy who not only has a "happy place," but also carries it with him wherever he goes, sharing joy and laughter with everyone around him and giving sweets to small children (cough).

Now that I've bleached my brain, I can answer a bit more honestly. Yes, in fact, there are good systems and I have seen good code, sometimes even great code, in my time. I would like to describe one such chunk of good code right now. Unfortunately, it will require a bit of background to explain what the code is, but please stick with me. Perhaps you can relax by going to your "happy place" first.

Questions for Kode Vicious? E-mail him at kv@acmqueue.com—if you dare! And if your letter appears in print, he may even send you a Queue coffee mug, if he's in the mood. And oh yeah, we edit letters for content, style, and for your own good!

One of my recent projects has been to extend support for something called hwpmc (hardware performance monitoring counters) on FreeBSD, the operating system I work on. As the name indicates, hwpmc is implemented in hardware, and in this case hardware means on the CPU. I don't know if you've ever read CPU or chip documentation, but there is little in our industry that is more difficult to write or less pleasant to read. It's bad enough that the subject is as dry as the surface of the moon, but it's much worse because the people who write such documentation either don't understand the technology or are terrible writers, and often there is a fatal combination of the two. Starting from that base, the typical software engineer produces code that somewhat mirrors the specification, and as things that grow in poison soil themselves become poison, the code is often as confusing as the specification.

What is hwpmc? It is a set of counters that reside on the CPU that can record various types of events of interest to engineers. If you want to know if your code is thrashing the L2 cache or if the compiler is generating suboptimal code that's messing up the pipeline, this is a system you want to use. Though these things may seem esoteric, if you're working on high-performance computing, they're vitally important. As you might imagine, such counters are CPU-specific, but not just by company, with Intel being different from AMD: even the model of CPU bears on the counters that are present, as well as how they are accessed.

The sections covering hwpmc in Intel's current manual, *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3B: System Programming Guide, Part 2*, encompass 249 pages: 81 to describe the various systems on various chips and 168 to cover all the counters you can use on the chips. That's a decent-size novel, but of course without the interesting story line. Kudos to Intel's tech writers, as this is not the worst chip manual I have ever read, but I would still rather have been reading something else. Once I had read through all of this background material, I was a bit worried about what I would see when I opened the file.

But I wasn't *too* worried, because I personally knew the programmer who wrote the code. He's a very diligent engineer who not only is a good coder but also can explain what he has done and why. When I told him that I would be trying to add more chip models to the system he wrote, he sent me a 1,300-word e-mail message detailing just how to add support for new chips and counters to the system.

What's so great about this software? Well, let's look at a few snippets of the code. It's important always to read the header files before the code, because header files are where the structures are defined. If the structures aren't defined in the header file, you're doomed from the start. Looking at the top of the very first header file I opened, we see the following code:

```
#define   PMC_VERSION_MAJOR   0x03
#define   PMC_VERSION_MINOR   0x00
#define   PMC_VERSION_PATCH   0x0000
```

Why do these lines indicate quality code to me? Is it the capitalization? Spacing? Use of tabs? No, of course not! It's the fact that there are version numbers. The engineer clearly knew his software would be modified not only by himself but also by others, and he has specifically allowed for that by having major, minor, and patch version numbers. Simple? Yes. Found often? No.

The next set of lines—and remember this is only the first file I opened—were also instructive:

```
/*
 * Kinds of CPUs known
 */

#define __PMC_CPUS()                                        \
        __PMC_CPU(AMD_K7,    "AMD K7")                      \
        __PMC_CPU(AMD_K8,    "AMD K8")                      \
        __PMC_CPU(INTEL_P5,  "Intel Pentium")               \
        __PMC_CPU(INTEL_P6,  "Intel Pentium Pro")           \
        __PMC_CPU(INTEL_CL,  "Intel Celeron")               \
        __PMC_CPU(INTEL_PII, "Intel Pentium II")            \
        __PMC_CPU(INTEL_PIII, "Intel Pentium III")          \
        __PMC_CPU(INTEL_PM,   "Intel Pentium M")            \
        __PMC_CPU(INTEL_PIV, "Intel Pentium IV")
```

Frequent readers of KV might think it was the comment that made me happy, but they would be wrong. It was the translation of constants into intelligible textual names. Nothing is more frustrating when working on a piece of software than having to remember yet another stupid, usually hex, constant. I am not impressed by programmers who can remember they numbered things from 0x100 and that 0x105 happens to be significant. Who cares? I don't. What I want is code that uses descriptive names. Also note the constants in the code aren't very long, but are just long enough to make it easy to know in the code which chip we're talking about.

Figure 1 shows another fine example from the header file. I've used this snippet so I can avoid including the whole file. Here, machine-dependent structures are separated from machine-independent structures. It would seem obvious that you want to separate the bits of data that are specific to a certain type of CPU or device from data that is independent, but what seems obvious is rarely done in practice. The fact that the engineer thought about which bits should go where indicates a high level of quality in the code. Read the descriptive comments for each element and the indication of where the proper types can be found, as in the case of pmd_cputype being from enum pmc_cputtype.

One final comment on this file. Note that the programmer is writing objects in C. Operating-system kernels and other low-level bits of code are still written in C, and though there are plenty of examples now of people trying to think differently in this respect (such as Apple's Mac OS X drivers being written in C++), low-level code will continue to be written in C. That does not mean programmers should stop using the lessons they learned about data encapsulation, but rather that it is important to do the right thing when possible. The structure listed here is an object. It has data and methods to act upon it. The BSD kernels have used this methodology for 20-plus years at this point, and it's a lesson that should be learned and remembered by others.

These are just a few examples from this code, but in file after file I have found the same level of quality, the same beautiful code. If you're truly interested in seeing what good code looks like, then I recommend you read the code yourself. If any place is a "happy place," it is in code such as this.

KV

P.S. Complete code cross-references for many operating-system kernels, including FreeBSD, can be found at http://fxr.watson.org/, and the code you're looking for can be found at http://fxr.watson.org/fxr/source/dev/hwpmc/. A similar set of cross-references can be found on the codespelunking site: http://www.codespelunking.org/freebsd-current/htags/.

Dear KV,

In his book *The Mythical Man-Month*, Frederick P. Brooks admonishes us with grandfatherly patience to plan to build a prototype—and to throw it away. You will, anyway.

At one point this resulted in a fad-of-the-year called prototyping (the programming methodology formerly known as trial and error), demonstrating that too little and too much are equally as bad.

What is your view of creating prototypes, and, particularly, how faithful does a prototype need to be to resolve the really tricky details, as opposed to just enabling the marketing department to get screen shots so they can strut the stuff?

Signed,
An (A)typical Engineer

Dear Atypical,

What do you mean by "formerly known as trial and error"!?! Are you telling me that this fad has died? As far as I can tell, it's alive and well, though perhaps many of its practitioners don't actually know their intellectual parentage. Actually, I suspect most of its practitioners can't spell *intellectual parentage*.

Alas, it is often the case that a piece of good advice is taken too far and becomes, for a time, a mantra. Anything repeated often enough seems to become truth. Mr. Brooks's advice, as I'm sure you know, was meant to overcome the "it must be perfect" mantra that is all too prevalent in computer science. The idea that you can know everything in the design stage is a fallacy that I think started with the mathematicians, who were the world's first programmers. If you spend your days looking at symbols on paper, and then only occasionally have to build those symbols into working systems, you rarely come to appreciate what happens when the beauty of your system meets the ugly reality that is hardware.

From that starting point, it's easy to see how programmers of the 1950s and 1960s would want to write everything down first. The problem is that a piece of paper is a very poor substitute for a computer. Paper doesn't have odd delays introduced by the speed of electrons in copper, the length of wires, or the speed of the drum (now disk, soon to be flash). Thus, it made perfect sense at the time to admonish people just to build the damned thing, no matter what it was, and then to take the lessons learned from the prototype and integrate them into the real system.

The increasing speeds of computers since that advice was first given have allowed people to build bigger, faster, and certainly more prototypes in the same amount of time that they could have built a single system in the past. The sufferers of prototypitis are really just chicken. Not putting a line in the sand is a sign of cowardice on the part of the engineer or team. "This is just a prototype" is too often used as an excuse to avoid looking at the hard problems in a system's design. In a way, such prototyping has become the exact opposite of what Mr. Brooks was trying to do. The point of a prototype is to find out where the hard problems are, and once they are identified, to make it possible to finish the whole system. It is not to give the marketing department something pretty to show potential customers—that's what paper napkins and lots of whiskey are for.

Where do I stand on prototypes? The same place that I stand on layering or the breaking down of systems into smaller and smaller objects. You should build only as many prototypes as are necessary to find and solve the hard problems that result from whatever you're trying to build. Anything else is just navel-gazing. Now, don't get me wrong, I like navel-gazing as much as the next guy—perhaps more—but what I do when I delve into my psychedelia collection has nothing, I assure you, to do with writing software.

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.