Beware of Programmers Carrying Screwdrivers

By Jack Ganssle
Embedded.com
(03/01/02, 02:24:36 PM EDT)

The earliest electronic computers were analog machines, really nothing more than a collection of operational amplifiers. "Programs" as we know them did not exist; instead, users created algorithms using arrays of electronic components placed into the amps' feedback loops. Programming literally meant rewiring the machine. Only electrical engineers understood enough of the circuitry to actually use these beasts.

In the 1940s the advent of the stored program digital computer transformed programs from circuits to bits saved on various media -- though nothing that would look familiar today! A more subtle benefit was a layer of abstraction between the programmer and the machine. The digital nature of the machines transcended electrical parameters; nothing existed other than a zero or a one. Computing was opened to a huge group of people well beyond just electrical engineers. Programming became its own discipline, with its own skills, none of which required the slightest knowledge of electronics and circuits.

Except in embedded systems.

Intel's 1971 invention of the microprocessor brought computers back to their roots. Suddenly computers cost little and were small enough to build into products. Despite cheap CPUs, though, keeping the overall system cost low became the new challenge for designers and programmer of these new smart products.

This is the last bastion of computing where the hardware and firmware together form a unified whole. It's often possible to

See 320+ hardware and software companies FREE! Embedded Systems Conference San Francisco, March 13-15. Get your FREE Pass today

reduce system costs by replacing components with smarter code, or to tune performance problems by taking the opposite tack. Our firmware works intimately with peripherals, sensors and a multitude of real-world phenomena.

Traditionally this close coupling of hardware and software meant that most embedded folks were EEs. CS majors generally didn't have the background needed to know exactly which bit to twiddle when. The typical documentation dearth meant we were all supposed to somehow figure out how things work just from the schematic. How many CS folks can do that?

But the world is changing. As the firmware grows and dwarfs hardware content perhaps EEs, especially those with the minimal software training most get, are not the ideal developers. I'm astonished that even today most EEs get essentially no training in their college years in real software engineering. Sure, they learn to code, but few graduate knowing much about configuration management, requirements analysis, methodologies, and all of the other aspects of the discipline.

Almost anyone can hack 10,000 lines of C into a reasonable product. That's quite impossible, though, when we're working on projects requiring hundreds of thousands of lines. Big code demands disciplined development, the sort that (I hope!) CS folks get, big time, in college.

Yet most firmware developers are EEs who drifted into software. That happened to me, though I've managed to keep feet in both hardware and software camps. How many of us EEs have the training for the huge projects that are coming along? My recent experiences teaching at the University of Maryland suggest that few do. Most new EEs still get zero exposure to true software engineering, which is astonishing when we realize that even hardware design in this world of ASICs, FPGAs, VHDL, and Verilog is an awful lot like building code. As usual, schools lag industry.

Computer Engineering is an intermediate approach that combines training in both hardware and software. I suspect that these folks may be the ideal candidates for future firmware development.

I love working at the margins of the code and the hardware, at making tradeoffs between programming and logic design, building ISRs and manipulating devices. The best projects control something moving, whether a robot or something less esoteric. Yet specialization is setting in. The future will find fewer developers tackling entire projects, and only a handful working with bits, bytes and soldering irons. I fear that unless EEs start getting a solid grounding in true software engineering, we'll be less valuable as developers.

What do you think? Are you an EE doing firmware? Did you get a solid background in disciplined software development, and if so, how?

Jack G. Ganssle is a lecturer and consultant on embedded

development issues. He conducts seminars on embedded systems and helps companies with their embedded challenges. He founded two companies specializing in embedded systems. Contact him at jack@ganssle.com. His website is http://www.ganssle.com/.

Take the poll

Reader Feedback

Good points, Jack! As a bonafide EE (BSEE and MSEE), I was once where many are today. However, the dark side beckoned, and I am now like you, living more in software than in hardware (including some projects using VHDL, which I agree is more software-ish than hardware-ish). I am amazed by how many EEs still feel that every clock cycle counts when they are programming for a 40 MHz RISC machine, eventhough the complete loop time (yes, many programs still have round robin loops for these large machines) is only using about 30% of throughput. Someday the benefits of reuse and components will catch up, and these EEs will be saying "What Happened?" as they look for new positions.

Joe Lemieux
Staff Engineer - System Architect
EDS

Interesting article!

I myself graduated an EE, then did firmware coding for a few years. The product I worked on was easily hundreds of thousands of lines. Like you observe in the article, I learned in school to code - not to participate in a very large software engineering effort. I learned largely by imitating what was in place, and reading a large amount of published material.

I'm not sure that any academic experience can truly prepare anyone for development in the corporate environment. There are two driving factors for this. The first is that the schedule cannot slip when finals are at a fixed time. You'd either have to accept a work in progress, or make something so simple that it couldn't go wrong. The second is that students must be isolated from each other's failures to a large degree. In the corporate environment, we all pick up the slack for underachievers, but this is frowned upon as "unfair" in the academic environment.

While I do think that CS majors may hear more about configuration management in overview form, I think that any company hiring recent grads should have a mentoring process that helps them to adopt a real engineering discipline.

I've since moved on to ASIC design, which is generally an even more unruly collection of EE recipients. Often, veterans in the field are only passingly familiar with source control, let alone configuration management. There is sometimes quite a challenge teaching software engineering methods to hardware hackers. I've known guys who will check out all their design files and keep them checked out indefinitely, making modification after modification, and finally dumping out dozens of unrelated changes days before a release

milestone. Thankfully, this is improving as HDL coders become more like software engineers than circuit jockeys!

How would you envision improving either academic training or early OTJ training to impart software engineering discipline to electrical engineers writing firmware or involved with HDL design and verification?

Anonymous

Hello. I read your article in Embedded Systems entitled "Beware of Programmers Carrying Screwdrivers", and it has a lot of truth, I'm afraid. I do a little programming in the industry, mostly because I work for a small company and have to wear a lot of hats. When I graduated with a BSEE three years ago, we did learn how to code a bit in C, Fortran, Pascal and Assembly, but we got zero training in software management. Everything we did was just to show that we could write code, and we huddled in groups to get something halfway coherent (sometimes that worked, but most often, it was mediocre at best.)

In fact, anyone who could code was practically worshipped and considered something of a prodigy because it meant that outside of the endless stream of homework, that person actually took time to learn a useful skill. To write code better was on everyone's lengthy list of goals, but few were able to achieve them. Just passing and keeping up were enough.

I was kind of lucky because in high school I had had some programming classes, but I don't consider them to have been great by any means, and I got by mainly because of a somewhat perfectionistic personality (i.e. write comments so that my grandma would be able to follow my programming logic.) I agree that there are a lot of holes in the college curriculum. (I had to teach myself C++ and AutoCAD.)

If I had to do it all again, the best advice I could give myself is to take more CS classes and get into the industry while in school so that the superficial examples that we were given as exercises have more meaning. Put in the time and learn as you go. Pay the piper and learn it right. Thanks for your insight.

Tom Barnum
Design Engineer
Communication Electronic Systems

First off, a correction.... all digital devices are also analogue devices. Ignore this at your peril.

Unlike most embedded folks I studied CS (ya know Fortran, Cobol, databases etc)twenty years ago on punch cards. I always had a low-level leaning even though I spent a while doing corporate databses and artificial intelligence for a while.

I self-trained in hardware design, though mainly digital. Pure analogue stuff still is a bit black magic to me.

If I look back at the years I spent in university, I reckon the only useful stuff I learnt outside the pub was algorithm

analysis; ie figuring out the pros and cons of using various algorithms and when to use which. IMHO this is one failing of the typical EE-writing-software. Too few EEs know eg that a bubble sort does not scale well or understand when to use a hashing table vs some other look-up strategy.

I reckon the embedded firmware enginnering area is the richest area in software development. It is also the most changeable. Stuff that you take as gospel today will likely be obsoleted within a few years. The most important attributes for people living in this zone are:

* keeping up to date.
* being able to assess the impact/opportunities of changes.

Charles Manning

We work on device drivers for ASICs used in multimedia applications. And we frequently have to work on evaluation boards. Generally, the experience has been that it is not necessary to know how the hardware works to debug your code - apart from rudimentary stuff like what jumper settings do what etc. However, if something like an EEPROM on the board goes bust, you generally call the hardware guy from the next cubicle!! By the way, I am an EE engineer.

Yashovardhana
Software Engineer
Philips Software

As a CS writing firmware, some of the very best code I have seen has been from EEs, typically those with a background in development of military products. This seems to imply greater understanding of the importance of a rigorous method and decent documentation. Some of the worst code I have seen is also from EEs, who don't seem to have the faintest understanding of how to layer code for abstraction and reuse, don't document at all, and have no conception of the value and costs of the most basic data structures.

Sean Kinghan

I totally agree with you Jack.

I graduated almost 4 years ago and I wasn't prepared to be an efficient embedded software developer. Upon graduation, I was very comfortable with assembler code (thanks to my senior project), but my experience with C and C++ was quite rudimentary. After I got my first job, I was instantly cast into the 32-bit PowerPC world and was more than overwhelmed. The complexity and steep learning curve of the development tools was a big enough obstacle, but throw in my little experience in higher-level languages and it only made matters worse.

I struggled at first, but I did learned C on the job. It wasn't pretty, but I was able to write code. As I learned more, I would go back and write portions of code over again to utilize new tricks I had learned. I received some well-needed training when my company sent me to the Embedded System Conference. I was exposed to what the rest of the

world was doing and it was an eye opener. I took a lot back with me from that trip and actually implemented some things I learned there.

My skills continued to improve, but I was nowhere near the developer I wanted to be. I finally lucked out because my project added a few outside consultants, who were computer science guys. I took one as my mentor and strived to learn everything I could from him. He taught me the ins and outs of Object Oriented programming and explained in detail the questions that needed to be asked when writing a software component. He also always stressed readability and making the code beautiful. I can hear him now, "White space! White space! Don't be afraid to use it!"

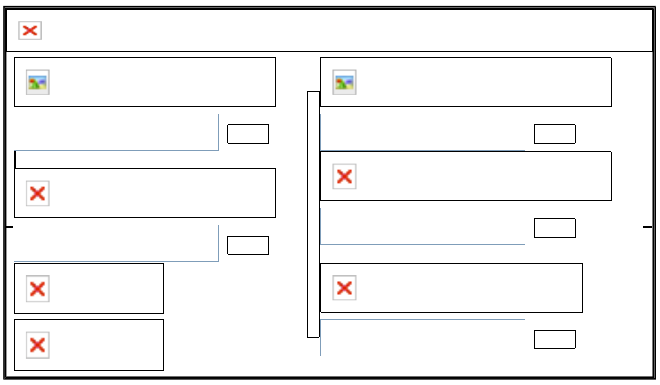
He also taught me the intricate details of the compiler, assembler, linker and the power of a good MAKE utility. Previously they had meant little to me, but now I'm more aware of their uses and I now know where I can make enhancements and get the most bang for the buck.

Without my mentor, I believe I would be half the developer I am today.

In conclusion, I learned the fundamentals in school with my EE education, but I had to learn how to write quality software from someone who knew software. I don't think there is any other way. I agree with you Jack, that the college ECE degree is the future of embedded programming. The code is getting too complex and it requires a good amount of experience to write code so it can be maintained in the future. The necessary skills just aren't a part of the EE curriculum today.

Sincerely,

Greg Hoge
Design Engineer
Liebert Corporation



Respond to this article:

Name: **(required)**
Title:
Company:
Email: