The Embedded Muse 167

Editor: Jack Ganssle (jack@ganssle.com) October 23, 2008

EDITOR: Jack Ganssle, jack@ganssle.com

CONTENTS:

****

This issue of The Embedded Muse is sponsored by Netrino.

Are you an embedded C expert? Know the difference between volatile `* const and const *` volatile? See how your skills measure up in a fun, quick, online quiz. Challenge your peers for bragging rights. All takers are automatically registered to win one of three iPod Nanos.

Start at http://www.netrino.com/Embedded-Systems/Embedded-C-Quiz-Muse

****

Editor's Notes

Did you know it IS possible to create accurate schedules? Or that most projects consume 50% of the development time in debug and test, and that it's not hard to slash that number drastically? Or that we know how to manage the quantitative relationship between complexity and bugs? Learn this and far more at my Better Firmware Faster class, presented at YOUR facility. See http://www.ganssle.com/classes.htm .

```
switch(city){
case San Jose:
```

```
case Phoenix:
case Austin:

Are you in the San Jose, Phoenix or Austin areas? I'll present a public
version of the Better Firmware Faster class in San Jose on December 8,
Phoenix on December 10 and Austin on December 12. Registration and
other info here: http://www.ganssle.com/classes.htm . You'll earn 0.7
Continuing Education Units, learn a lot, and have more than a little
fun. Sign up before November 8 and receive a $50.00 discount.
}
```

Harold Teague has an interesting question for Muse readers: We're in the process of establishing job roles along the lines of firmware architect, firmware design engineer, and firmware integration/systems engineer. Does anyone have these or similar descriptions already defined?

Let me know what you think and I'll publish the results here.

In the seemingly never-ending quest for the perfect communications monitor, Bill Eubank sent info about an RS-232 monitor that runs on a Game Boy! See http://www.databoy.netfirms.com/. He writes: I have had one of these for may years and I still use it regularly. It monitors both sides of the communications, receive and transmit. It can also capture this information and transfer it to a PC for storage and/or printing. I met Yosuf Taraki, the designer, several years ago. He's a very nice guy.

Book Review

After the last Great Book Giveaway, J. Silva submitted a review of "Practical Software Estimation":

Function point estimation methods are usually glossed over in a small section in program management books. "Practical Software Estimation" by M. A. Parthasarathy provides an in-depth discussion of function point-based methodologies for program management.

The author gives a thorough explanation of the process with a few examples for each definition and case studies throughout. The first few chapters of the book are dedicated to introducing the process and basic concepts. The main meat of the material is another four chapters. The last half touches broadly on how to use the process for various types of projects (new development, migration or maintenance), as well as addressing metrics and estimation methods. It has some useful information for outsourcing project, which can shortcut lack of experience.

Unfortunately, with all these helpful examples, it was difficult to synthesize much of the information into something useful in real life for a number of reasons. First, the process seems so complicated that it is difficult to apply it to some projects. From the high level, the process is simple but the devil is in the details. Second, it seems time consuming. I do not think this method would work for small, fast projects.

Third, it requires the project manager to not only have a strong understanding of the requirements, but also how the design would breakdown. It is difficult to discern if this is a fault with the actual method or the author's presentation.

I would recommend this book to a program manager on a large-scale, complex government contract with a long time horizon and a need for accurate schedule and costs. I would not recommend this for startup organizations, smaller project developers, or novice program managers.

The Failure of Reuse

Reuse is the holy grail of software engineering, one that is so entrenched in our belief system no one dares to question its virtue. The quest for reusable components is one of the foundations of object oriented programming and all of the tools and languages that OOP has spawned.

Yet, my observations suggest that at least in the embedded space reuse has been a dismal failure.

First, let me define "reuse." We can talk about carrying-over code, or salvaged code, both of which imply grabbing big source files and beating them into submission till a new product appears. Not to knock that; it's much better than starting from scratch each time (but see my comments towards the end). We reach the nirvana of reuse, though, when we leave the source code alone, when we're able to pluck a module from the virtual shelf and drop it in to a new project, unchanged.

Martin Griss and others have observed that a module isn't really reusable till it has been reused three times. No matter how good our intentions, the first time we try to reuse something we discover a facet of the new problem the old module just can't manage. So we tune it. This happens a couple of times till the thing is generally reusable. That's not because we're stupid; it's simply because domain analysis is hard. No one is smart enough to understand how a function might get used in other apps.

It's expensive to do a forward-looking design of a function or module. You'll always save money –in the short term – solving today's very specific problem while ignoring the anticipated demands of future projects. If you elect to pursue a careful program of reuse your projects will initially come in late and over-budget.

Reuse is hard. It's like a savings account. My kids complain that if they stick a few bucks in the bank then that's money they can't use… and it's just a piddling sum anyway. Sure. The value of savings comes after making regular deposits. Ditto for reuse. The cost is all up-front, the benefits come from withdrawals made in later years.

Sure, we all know the long-term outweighs today's concerns. (Though the recent financial meltdown suggests that's a lesson some CEOs have missed). Most bosses buy into the idea of the benefits of reuse, without being willing to stand the pain of creating the reusable components. And that's the rub. When the ship date looms closer, too many bosses will tell us to toss out any sort of discipline that has long-term benefits in pursuit of a near-term release. So, in practice, reuse often fails since schedules generally dominate over any other parameter.

I also suspect most of us don't want to reuse code. 35% of RTOSes are homebrew, despite at least 100 commercial – free and otherwise – products on the market. Nothing is easier to beg, borrow or buy than an RTOS, yet most of us still refuse to practice even this most simple of all reuse strategies. Why is this? I've heard many specious arguments (too expensive, yet there are plenty of freebies; we don't trust the code, but some are safety certified, etc), plus a few really good arguments (it's all legacy code, no one is willing to make risky changes).

I think aggressive reuse is our only hope of salvation from the morass of expensive and unreliable code. Building correct firmware is so difficult that we and our bosses have a fiduciary responsibility to find ways to make it reusable.

But we're not doing it. Sure it's hard. Yes, it's initially expensive. And of course we cannot reuse everything; a lot of what we build will always be inherently unreusable, like hardware drivers that get tossed with each new spin of the design.

So what's the deal? Why is reuse such a failure? Is the best we can do, or chose to do, is to salvage old source files?

A closing comment: In "Empirically analyzing software reuse in a production environment," Selby et al showed that hacking source has quickly-diminishing benefits. Once one changes more than about a quarter of the code there's not much cost saving over starting from scratch.

Jobs!

Let me know if you're hiring firmware or embedded designers. No recruiters please, and I reserve the right to edit ads to fit the format and intents of this newsletter. Please keep it to 100 words.

Netrino has multiple open positions for embedded software developers and FPGA and board-level electronics designers, in Maryland and California. Join the Embedded Systems Experts(tm) in a fun work environment that offers challenge and variety. Learn more at http://www.netrino.com/Embedded-Systems/Embedded-Software-Jobs

Watt Stopper/Legrand has 3 openings for Electronic Design Engineers and one opening for an Electrical Analog Design Engineer in Carlsbad, CA. Products include occupancy sensing wall switches, home lighting control devices, commercial lighting control panels, and day lighting control products. Apply for these jobs online: http://jobs-legrand.icims.com/jobs/intro

Are you an experienced Embedded Software Engineer? Do you envy people whose work doesn't require them to live close to a major city, who have a five minute commute to the office, and who have a ski resort, world class fishing, and even a glacier minutes from their home? If so, check out Pacific Insight Electronics Corp. in Nelson, BC (pacificinsight.com). We are looking for an Embedded Software Engineer. If you are driven to build high-reliability, automotive-quality software, and you dream of doing this in land of lakes, mountains and eagles, give us a call. We need someone who is bright and motivated, with solid experience writing top-quality C code for real-world applications. Experience with LIN, CAN, instrumentation and software modeling would be a bonus, but mostly we want someone who learns quickly, who has proven C experience, and is able to work well in a team. Of course, we need you to have an Engineering degree and preference will be given to candidates who are registered as a Professional Engineer. Contact Amanda at alaughton@pacificinsight.com.

Joke for the Week

Guillermo Zepeda-Lopez sent this:
There is an effort in our company to find ways to make our automotive ECUs lighter. Two of my SW colleagues (Ricardo Garcia and Antonio Lozano) turned in some great ideas to reduce weight through the SW in our product:

- Write "lighter" code.
- Declare all variables as "volatile".
- Recursive calls to function ReduceWeight(int CurrentWeight).

- Use more powerful microcontrollers so that we reduce the CPU "load".
- Delete all comments from the code.
- Use an operating system with a bigger "footprint". That way, we distribute the weight on a bigger area (less pounds per inch).
- Leave all I/O pins "floating".
- Always use "Tiny" memory model.
- Stop using processes. Use only "light-weight processes".
- Do more "on-the-fly" processing.
- Stop drawing state diagrams with rectangles, use "bubbles" instead.
- Don't eliminate all bugs. Remember some bugs can fly.

About The Embedded Muse

The Embedded Muse is an occasional newsletter sent via email by Jack Ganssle. Send complaints, comments, and contributions to him at jack@ganssle.com.

The Embedded Muse is supported by The Ganssle Group, whose mission is to help embedded folks get better products to market faster. We offer seminars at your site offering hard-hitting ideas - and action - you can take now to ***improve firmware quality and decrease development time***. Contact us at info@ganssle.com for more information.