The Embedded Muse 161

Editor: Jack Ganssle (jack@ganssle.com) June 10, 2008

EDITOR: Jack Ganssle, jack@ganssle.com

CONTENTS:

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 .

Contest Results

Microchip gave me an MPLAB starter kit as a giveaway, and in the last Muse I offered it as a contest prize. The question was "what will embedded development be like in 10 years."

Lot of interesting responses arrived, including a complete original SF story that I can't reprint here as it has now been submitted to a magazine. The winner was Ignacio Gonzalez Torquemada, who wrote: In 10 years, everybody will have from 2 to 26 fingers, depending on his needs, and so, at last, we'll get rid of base ten numbers, and revert to the more useful base sixty (babylonian type), so I suppose that, when you wrote "10 years",

you were actually meaning "10 base sixty years", that is, sixty years, well, 60 base ten, you know.

So, in 60 years (that is, in a period of time equal to 98% of comet Westphal 1's orbital period, to avoid ambiguity) I predict the following:

1) I'll be still reading, hearing, seeing, or just getting, The Embedded Muse.

2) Embedded development will be all about governing nano-robots with your mind. Your boss will come, tell you what he thinks the customer thinks he wants, you'll think the problem in terms of armies of nano-robots, the nano-robots will appear, build another nano-robots, build the necessary electronics, networks, middleware, software, re-configure themselves, test themselves, deploy themselves and solve the problem; your boss will come and tell you that you have solved the wrong problem, and... well, as usual.

3) Intel will have disappeared. Ditto Freescale, Texas, Philips and ARM. The pervasive nano-robots will be programmed in Chinese, and the thingware behind the scenes (where thingware will be a mix of hardware, software, middleware and underware) will be produced by Yagooglehoo Inc.

Abdallah Ismail wrote: Graphical-based development could become dominant, especially but not limited to the FPGA market...Cypress' Design Express, NI's Labview, Altium's Designer, and Simulink are just few examples fueling this development stream...the tool will become in charge of the development process, not the programmer :(

DSProcessors will become obsolete, or at least the term will, overtaken by the ever-improving easy-to-use microcontroller...The DsPIC driving this very contest is a glimpse of the future of microcontrollers (and DSProcessors?) to come.

32-bitters will takeover...For example Luminary's ARM portfolio include 28-pin microcontrollers within the 2$ range...They're strongly pushing for 8 and 16-bitter replacement, and with very convincing arguments I must say!

Governed by the theory of evolution, the industry will converge more and more towards few standard architectures, with few standard compilers, all the open-source code in the world, and 10$ development kits :)

This from Donald L. Dobbs: Embedded development will have bigger, better, faster computers for simulations and cross-compilers. The software tools will be more powerful. These changes are just a part of the natural progression of technology.

What will not change is the quality of the code. As the general said: "those ignorant of history are doomed to repeat its mistakes." Each generation of newbies will write the same sloppy code because they won't stand on the shoulders of those who preceded them. Managers, in particular, are culpable because they never seem to know enough about the technical aspects of good code development to either instill or insist on the training and methodologies needed to produce solid code. The Carnegie-Mellon stuff is focused on big systems. I doubt if any managers or programmers in embedded shops are even aware of C-M unless the shop is part of a bigger organization forced to embrace C-M methodologies (requirements).

Sorry to sound so negative, but I've been involved in SW dev. since 1960 and nothing seems to change. C & C++ code is even buggier than the code we wrote using Fortran, PL/I and Cobol. Languages come and go and really have nothing to do with the quality of the code. Skilled programmers can write good code in any language handed to them; bad programmers can even screw up Ada.

Firmware – Best Practices

In response to Richard Wall's request about teaching best practices readers had a lot to say. Gary Stringham wrote: A required part of developing and maintaining quality firmware (a.k.a. embedded software) is to have proper hardware support. I have seen many examples of poor-quality firmware because of constraints imposed by hardware that should not have been there. To that end, I have taken on the mission of promoting best practices in hardware/firmware interface design. See www.garystringham.com/newsletter for several such best practices.

You have asked more particular about best practices for firmware quality. So here are a few that I have used in my firmware development. Some of them may be non-conventional but they have served me well.

- Collaborate with Hardware Engineers: If possible get acquainted and establish a dialog with the hardware engineers about the chips you are using. This will help you have a better understanding of the chips and help them to understand firmware so they can design better chips.

- Stay in the Same Job: Stay in the same job for at least two versions of the same product. During the first version, you are learning your job. In subsequent versions, you know how to do it you can work on improving it. You will see what porting and maintenance issues pop up which you can then address.

- Work Yourself Out of a Job: Make the code so that very little effort is required to port it

to the next version of the product. Implicit in this is that the code is of high-enough quality that very little debugging effort is required to get it going on the new product. Working yourself out of a job will free up time to take on and improve new tasks.

- Provide a Logging and Dumping Facility: Put in a simple and lightweight logging and dumping facility in the code and leave it there. It will prove useful in the future when integrating the whole system. Have it be able to dump a ring buffer of events, current values of variables and data structures, and current values of pertinent hardware registers.

- Test the Debug Code: The debugging code that is left in should undergo nearly the same level of quality testing as the main code. Otherwise, you will waste time trying to debug the main code when the defect is in the debug code.

- Test Using Hackware: Put temporary hacks in your modules to induce various conditions for testing. Use hacks to induce all flavors of errors and boundary conditions within the module, to pretend that operating system calls return errors and bad results, and to pretend that the hardware registers are reporting errors and difficult-to-reproduce conditions. When done, take out the hackware. Although hackware cannot be used in automated and regression test suites, they serve a useful purpose if designing, creating, debugging, and maintaining such in-depth automated regression test suites are cost prohibitive.

- Be Able to Debug a Buttoned-up System: System-level problems pop up when real production hardware is running real production firmware under typical and heavy loads. But at that point, in-circuit emulators, JTAG sniffers, debug monitors, and other external probes can no longer be attached. Provide some back-door method to get at the code and dumping facility.

These best practices helped me to produce quality code that set a standard for others to follow. I never got the award for most defects fixed. One engineer jumped for joy when he finally found a problem that was due to a bug in my code when most of the time the bug was in his code.

Here’s John Carter’s take: Sigh! Those lists will produce neat, stiff and wrong code.

Two items must be added.

1. Take the time to understand Design by Contract, in particular the notion of class invariants. Design your classes so their invariant cannot be smashed by any means.

If you don't believe me, believe Bjarne... http://www.artima.com/intv/goldilocks3.html

2. Reduce coupling, increase cohesion.

Most else follows from those two principles.

I have to agree with John. Design by Contract is hardly rocket science – it's similar to the age-old idea of checking your goesintos and goesoutas – but can lead to brilliant results. Here's a three-part series I wrote about it:
http://www.embedded.com/columns/breakpoint/197008824
http://www.embedded.com/columns/breakpoint/198701649
http://www.embedded.com/columns/breakpoint/199202728

Bill Stefanuk wrote: Like Richard, I too teach embedded system and software development. Here's my pet peeve with students (and some others I've worked with along the way) that could be rolled into Richard's list. It may seem totally obvious but, based on the number of students that I've seen go down this road, apparently its not.

Zero tolerance for ***WARNINGS***

I'm going to make it a rule after this past semester that I won't help students with their problem until they can show me that their code compiles and links without warnings. Amazing how many problems go away when the warnings do, and how many undiscovered ones won't need to be found and fixed.

From Peter Miller: This omits the most important ones:
* write tests
* require, no, _mandate_ tests
* reward testing

> 2. Impediments to quality code

Number One Worst Problem: The "you can't test embedded code" assertion.

I can't tell you how often I've heard this, and it's plain wrong.

Some of the many many ways to test embedded code include:

- If it's pure assembler, run it in an emulator. Not using one (software or hardware) is a false economy. Test your code on the emulator before you load it into the hardware, preferably scripted for repeatability.

- Find ways to mimic your hardware's environment (using as much "real" code as

possible), especially all the unlikely corner cases. Try some "impossible" cases as well. The routine every day stuff is largely uninformative for testing.

- In addition to cross compiling, native compile your code on your build host, and test each module, each function, each class *on *the *build *host*, before you ever load it into your development prototype.

- You should be delivering *at least* as many lines of test code as you deliver product.

- Tests capture how the software is *supposed* to work, in a way that plain code cannot (even literate programming, of which I a big fan). Tests represent an important form of "corporate memory", allowing your project to move between departments, between contractors, between processors, between compilers.

Thad Smith contributed: My personal item, often not high on other people's list is to document interfaces accurately and in detail. Many interfaces are function calls, so a lot of focus goes into documenting those well. Covered are fundamental description, what must be true prior to calling the function, what is true on return, and detailed description of parameters, including units of measurement, special case values, and boundary values. My own style is to place all information needed to call modules functions in the C header file associated with the code file. If you have to look at the implementation code to determine the answer to a question concerning usage, the interface probably isn't defined well enough. Having a well defined interface helps allows the calling code and called code to be independently checked against the declared interface.

In C, typedefs are a convenient shortcut to assign attributes to several variables of the same type. Document the units, range, and meaning of the type with a typedef. Variables declared with the typedefed type inherit all the meanings for the type, reducing the need to reiterate units, ranges, etc. They also promote conceptual thinking, in my opinion.

Cliff Wignell enjoyed the reference to Zen and the Art of Motorcycle Maintenance and wrote: I just wanted to mention that your comment about Robert Pirsig (Firmware- Best Practices) made me smile, my High School English teacher told me I should read it, she even presented a copy to me, it took me 7 years to get around to reading it, I ended up reading it three times, the message did not begin to appear until the second reading, a case of concentrating on the words not the message.

Software development is both an Art and a Science, the Science part deals with the methods, algorithms, timing, bits and bytes; the Art it choosing which parallel path, function naming and formatting etc. I think the Science is the easier part to teach, with

the art you can provide the rules and suggestions but the student needs to decide to own it, read it more than once.

My favourite language is C, it to me is a natural extension (abstraction) of the processor; well written C is a delight, with the correct selection of naming it will read almost like English.

It becomes, I think pride in your work and the greatest complement you can receive is during a code review, no comments are made or someone states that is loverly.

Many I think see the act of coding as the penultimate act; I do not think it is, it is the mechanical process of transcribing the Science and Art into a machine readable form, a process not the end in its self; if that makes sense.

I use a PCode approach, (as per writing Solid Code MS Press) so my code is in effect documentation with code inserted.

One thing I do, is when I am writing and I am about to use a function for the first time (especially with MS products) I write a little test program to all the function so I can check that I properly understand the parameters and returns. I write a lot in Windows (& Linux) and I am still surprised by functions that do not completely operate as described, these bugs are very difficult to find.

Last comment, is when I write a function, especially the larger one, similar to the above I will write a test program to test it and make sure I have not done any thing stupid.

Tony Gray wrote: In response to your request for comments on best practices, Id like to add the following:

Keep your variable and function names in synch with their usage. As a project progresses its not unusual to find that a particular variable or function ends up in a slightly different role than you originally planned. There's nothing wrong with that, but you must update the name to match the new usage. I've seen code reviews get derailed into a ten minute discussion of the role of a particular variable, only to find that the confusion stems from a mismatch between the name and the usage.

Id also add that in my opinion Code Complete is the bible of best practices and should be part of every university's software curriculum.

Chris Nelson sent this: I've long thought that if I ever had a chance to design a software engineering curriculum, I'd include a mandatory, two-course sequence that was only

taught in Fall or Spring semesters so you had to take months off between the courses. The twist is that the second course is entirely maintaining and enhancing the code you produced in the first course. Or maybe half maintaining your own code and half someone else's. In the confines of a one-semester course, you could have project N be changes to project 1 (where projects 2, 3, ... N-1 had nothing to do with that). I'd also pair off students in user/developer pairs and let the user clarify for the developer any ambiguities in the project spec, perhaps by editing the spec so that the grader could compare both products.

John Davies, too, had some ideas on how to teach writing quality code: I majored in EE but took as many CS classes as I could. By accident, I took a CS class in List Processing Programming. I was the only EE in the class. I should have dropped it immediately because I was way over my head in the class. I had the bare minimum requirements to take the class, but the next year the professor was planning on raising the requirements. He decided that since most of the class met the raised requirements he would teach as if we all did.

However, the last part of the class was going to be about artificial intelligence. It was 1980 and what could be cooler than AI? So I stayed in.

First class, he said that although we could write our assignments in Fortran, Pascal was preferred. So I went to the bookstore and bought the only Pascal book they had. It sure wasn't _Pascal for Dummies_ or _Learn Pascal in 21 Days_ back in 1980. It was more a book on Pascal syntax on the off-chance I wanted to write a Pascal compiler.

The professor did two very good things in the class.
1. The assignments built on each other. So the goal wasn't to get the assignment done then throw it away. It was my first exposure to eating my own dog food.
2. After each assignment, he provided his own well-written, clean code. We were allowed to either use his code or our code for the next assignment.

My first program was Fortran-ish code written in Pascal. Part of it might have even worked. The next program was leveraged off his code and worked much better. It also looked more like Pascal. Eventually I was confident enough to start eating my own dog food.

I got a B out of that class but it was tense for awhile.

Troubleshooting

Tony Gray wanted to add onto the on-going debug discussion: Mr. Pelc is spot on with his formal scientific method approach to debugging. The one thing I would add is to start at one end of the cause-effect chain and work your way to the problem. For example, lets say you're debugging a routine that converts an A-to-D count from a thermistor into a temperature and that, to start with, you have very little information other than the temperature your software calculates is incorrect. For this example the chain of cause-and-effect starts with the sensor itself and ends with the return value of your function. Begin at one end of the chain and move forward or backward. In this example you could start by measuring the voltage across the sensor, then verify that the conditioned voltage on the A-to-D pin is correct, then verify that the converted A-to-D value is correct, and so on. Or you could start at the other end and work backwards starting with the last step of the conversion algorithm. Too many times we engineers jump into the middle of a problem, making assumptions that what came before or what comes after must be correct. Verify those assumptions before forming your hypothesis.

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.

Draper Laboratory is an independent, not-for-profit engineering research and development organization. Its mission is to serve the national interest in applied research, engineering development, education and technology transfer. The core competencies leveraged in Draper's work are guidance, navigation and control for strategic systems; precision targeting and information management and decision systems; autonomous air, land, sea and space systems; reliable, fault-tolerant embedded software; and miniature, low-power electronic and mechanical systems. We are located in Cambridge, Ma.

We have an opening for a Software Engineer with 3 years embedded software development skills using C. Projects include working on micro-electronics systems, soldier portable systems and unmanned air and ground vehicles. Specific exposure to any of the following is a plus: PowerPC, TI DSP, GPS, audio signal processing.

Relocation assistance is available.

To apply send a resume to nbeaumont@draper.com

Lutron Electronics Inc., located in the Lehigh Valley PA area, is the world leader in the design and manufacture of lighting controls systems and shading solutions. Currently we are seeking Embedded Design Engineers. You will work on requirement gathering, architecting scalable systems, user interface design, databases, development, and testing. Also, you will assure high quality standards, good coding practices, process improvements, clean designs, smart testing and review practices. Qualifications are Computer Electrical Engineering degree, 3+ yrs. experience in software/hardware development, C++ object oriented programming, and familiarity with real time operating systems. RS485 or Ethernet experience is a plus. Apply on-line at www.lutron.com Careers Section.

Z Corporation (Burlington, MA) designs, manufactures, and sells the worlds fastest 3D Printers, machines capable of building three-dimensional models directly from digital data. We are seeking a software, computer, or electrical engineer to join our dynamic team in co-developing the hardware and firmware aspects of control for our 3D printers. As part of a multidisciplinary design team, this engineer will be responsible for programming in C and C++ interfacing with programmable logic, microprocessors and analog circuitry, as well as system integration.

The ideal candidate will have significant product design experience with C and C++ for hardware/software integration of embedded systems using real time operating systems, have experience interfacing real world electromechanical devices, and be familiar with logic design using FPGA's. Candidates should have an engineering degree (computer science, computer engineering, or electrical engineering), five years or more experience, strong academic and work references, and be excellent communicators.

Requirements/duties:

- BSEE, BSCS, BSCE or equivalent
- Proven object-oriented design skills in C++ utilizing inheritance and exception handling.
- 5 years or more product development experience
- Multi-threaded programming for x86 or other 32 bit processor using a commercial embedded real time operating system.
- Design skills; software & hardware codesign.
- Documentation: writing specifications, test benches and documentation of finished code.
- Continued maintenance & support of released code.
- Used to working alone or in a small team.
- Motion control; electromechanical interface.
- Familiarity with SPI, I2C, and other standard communication protocols.
- Programming 16 bit MCUs in C.

- Understanding of HDLs like Verilog for PLD or FPGA design.

Please send resumes to recruiting@zcorp.com

Buglabs is searching for an Embedded Developer. Ideally based out of New York - we are looking for passionate, experienced Device Driver and Linux Developers - we are talking about the kernel, driver internals, ARM experience, and the Linux bus architecture. You use and contribute to Open Source Software, work with cross-compilation and embedded tools, C programming, and shell scripting. You are also familiar with several desktop and embedded distributions. Getting your hands dirty and working deep in the code is what you love most.

The posting is here with more details (http://www.buglabs.net/jobs#embedded_linux_developer) and below.

Major Responsibilities
* Write, maintain, and port Linux device drivers
* Manage and update make files and build scripts
* Make strategic choices on software packages and open source groups to utilize and contribute to

Minimum Requirements
* Proficient in 'C' programming
* GNU tools/Cross-compiling experience
* At least 2 years Linux device driver development experience
* Linux 2.6 kernel Kconfig/makefile familiarity
* Experience with Wifi, I2S/audio and power management drivers
* Linux application development experience
* Low-level hardware device experience
* Exposure to full software development lifecycle
* Embedded Linux development experience
* Distributed application development experience
* ARM familiarity
* Embedded system design experience
* Experience and participation with open source project(s) is a big big plus
* Self starter - ability to work independently as well as in teams
* Demonstrable background in computer science (algorithms, time/ space complexity, abstract datatypes)
* Excellent communication skills a must

Joke for the Week

Frequent contributor Catherine French sent these Haikus for Windows:

The Web site you seek
Cannot be located, but
Countless more exist.

Chaos reigns within.
Reflect, repent, and reboot.
Order shall return.

Program aborting.
Close all that you have worked on.
You ask far too much.

Windows NT crashed.
I am the Blue Screen of Death.
No one hears your screams.

Yesterday it worked.
Today it is not working.
Windows is like that.

Your file was so big.
It might be very useful.
But now it is gone.

Stay the patient course.
Of little worth is your ire.
The network is down.

A crash reduces
Your expensive computer
To a simple stone.

Three things are certain:
Death, taxes and lost data.
Guess which has occurred.

You step in the stream,
But the water has moved on.
This page is not here.

Out of memory.
We wish to hold the whole sky,
But we never will.

Having been erased,
The document you're seeking
Must now be retyped.

Serious error.
All shortcuts have disappeared.
Screen. Mind. Both are blank.

I ate your Web page.
Forgive me; it was tasty
And tart on my tongue.

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.