The Embedded Muse 172

Copyright 2009 TGG January 5, 2009

EDITOR: Jack Ganssle (jack@ganssle.com)

CONTENTS:

****

This issue of The Embedded Muse is sponsored by Smart Bear Software (http://smartbear.com/?EM) .

Complimentary Books (http://codereviewbook.com/?EM) for you and your team...
Get your complimentary copy of our book, Best Kept Secrets of Peer Code Review. "A very well-written 164 page book that's a fascinating read. The benefits of inspection are so profound that even the smallest outfits must take advantage of this technique." -Jack Ganssle

Order your complimentary book today! http://codereviewbook.com?EM

****

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

Brazil! The country of samba, feijoada and caipirinha. And, of course, embedded systems. I'll present a two-day version of my Better Firmware Faster seminar in Sao Paulo March 25th and 26th. See http://workshop.embarcados.com.br for more information.

Most of us have heard about the Zune debacle by now. It's astonishing that we're still making leap year errors! Bob Paddock's blog has a good description of what happened: http://softwaresafety.blogspot.com/ .

Larry Ruane has developed an open source protothread manager that combines event-driven techniques with threads. Check out http://protothread.wiki.sourceforge.net .

Quotes and Thoughts

Geoff Patch sent in this one: The process of preparing programs for a digital computer is especially attractive, not only because it can be economically and scientifically rewarding, but also because it can be an aesthetic experience much like composing poetry or music. - Donald Knuth

Response to Last Issue's Quotes and Thoughts

> We know about as much about software quality problems as they knew
> about the Black Plague in the 1600s. We've seen the victims' agonies
> and helped burn the corpses. We don't know what causes it; we don't
> really know if there is only one disease. We just suffer - and keep
> pouring our sewage into our water supply.

Steve Litt disagreed: I disagree with this entirely. On one level, we know exactly what causes bugs:

* Uninitialized variables and pointers

* Overrunning arrays
* Aiming pointers at the wrong place
* Using local strings as function returns
* Picket fence conditions
* Etcetera

Perhaps he meant we don't know how it could be solved. Again, I at least partially disagree. If programmers had the time and compensation to code without bugs, they could:

* Code modularly
* Document each subroutine: input, output and intentional side effects
* Use proper and consistent naming and indentation
* No hotdog coding. One-liner loops aren't a badge of coolness.
* Build a test jig for each subroutine and test under varying conditions
* Test like that at each level up the tree, up and up to the top
* gcc -Wall, then fix every single warning
* Get lint and use it
* Get and use valgrind or another leak finder
* Get and use a memory tromp finder (maybe run it under VMS :-)

3RD PARTY LIBRARY DEFENSIVE CODING
* If it isn't exquisitely documented, don't use it.
* If its source isn't available, be very afraid
* But don't change its source unless absolutely necessary
* Run valgrind or another leak finder on it first, dump it if it leaks or doesn't give back at the end and the vendor can't give a good explanation
* Run a memory tromp finder, dump it there are memory tromps the vendor can't quickly fix or give a good explanation for.
* Build several test jigs for it, and put it through its paces. Dump it if the vendor can't explain and ameliorate every failure.

Of course, we don't have time for this. Time to market is king. The company who actually did this might get scooped every time and go out of business, or at least that's the prevailing thought. The coder who does this will be slower (according to prevailing thought) and will get fired.

Meanwhile, the customers prioritize newer, cheaper, sooner over bug free. It's not much different from people enduring the hassle of shopping at Wal-Mart because they save money.

So bottom line, the human cause of bugs is nobody is willing to pay to keep them out.

Interesting Survey

Paul Bennett wrote in about a survey of the embedded space: Some of your readers may be interested in this survey: http://www.esemagazine.com/index.php?option=com_content&task=view&id=529&Itemid=4 (WARNING from Jack: Firefox claims this site has been reported to contain malware; however, Google's inspection of it reveals no problems over the last 90 days.)

It shows a couple of interesting trends.

Of note is that, on average, the software development teams have increased by almost two people while they hardware development teams have remained static. The article makes no mention of those developers who do both hardware and software.

This is of particular interest:- "The overwhelming favored tools are the compiler/assembler and the debugger, with the oscilloscope a distant third. It's also an interesting anomaly that software testing tools are way at the bottom of the list, yet users say that "test" in general is one of the key factors in the design process often taking the majority of the design time. That leaves me to conclude that users aren't happy with their current software test tools."

I can understand why the software testing tools remain a low priority, especially in the true embedded systems market. Simpler and more direct techniques can often yield quicker and more definite results. I rarely use anything more complex than a DMM, Logic Probe and oscilloscope with my Forth tools for debugging hardware and software.

Good news is that software re-use is alive and well according to the survey. That is re-use of in-house developed software.

However, the really interesting item was this.

"Another response that I have a tough time justifying is that developers this year consider the "chip itself" to be more important than the ecosystem surrounding the chip (such as software, tools, and support). I constantly harp on the processor vendors how important it is to have their ecosystem in place, and that's the only real road to success. System developers seem to think otherwise.

Also: "The number of developers that don't use programmable logic in their designs stands at 52%, a relatively (and surprisingly) high number. When we asked them why, the top answers were that programmable logic is too expensive, consumes too much power, and is too hard to use. The vendors I spoke to all refuted these claims, but it appears that that message is not getting out."

I am wondering whether or not the reason that this seems to be the case is that with some of the newer low power processors being used, the interfaces can be simplified to use really low power chips by letting the processor take up more of the slack. This may get to be more the case if the new low power multi-core processors truly embrace the mesh processing models where you can ascribe different functionality to individual cores and make the information flow accordingly.

I know you helped the article authors with some of the data-points and I have not raised comment on that section.

Priceless Datasheets

Thilo Lauer contributed this fun datasheet: Today i'd like to share a short quote of the SH7125 Hardware Manual from Renesas. While their overall structure of their web appearance is hopefully not to be compared with what they built into silicon, I found this sentence on the bottom of page 21 answering all my questions:

"The flash memory can be programmed with a programmer that supports programming of this LSI, [...]"

Comments on My Microchip Comments

Jon Marsh wrote: I wanted to make a couple of points on your 8-bit vs. 32-bit comments in the newsletter last week. I should probably start by declaring my bias in that a majority of my consulting work at the moment is migrating 8051 designs to Cortex-M3.

There's no doubt that 32 bit prices will continue to fall. Consider Luminary Micro's ARM controllers. They, like ST, Atmel and others, are doing a remarkable job of positioning the ARM as the new century's 8051. Costs keep coming down; some of these parts are available for under a buck.

I think we are close to that point already.

The industry shipped over a billion ARM processors in the past quarter, giving nearly 4 billion in 2008: http://www.arm.com/news/23608.html .

Microchip shipped just under a billion PICs in a year:
http://www.microchip.com/stellent/idcplg?IdcService=SS_GET_PAGE&nodeId=2018&mcparam=en534302 .

It's hard to get total annual figures for 8051 because there's no central "owner" of the architecture keeping count, but it wouldn't surprise me if it had already been overtaken by the ARM in terms of volumes.

>Excluding the hopefully short-term fallout from the financial
>mess currently rocking the world, I think not. "Transistors are
>free" is our mantra, but of course that's not really true.
>Transistors do have a cost, both in dollars (well, microcents),
>size, and power consumption. With similar technology 8 bit
>controllers will always outperform a larger CPU in these categories.

While I totally agree that transistors aren't free and die size always matters when it comes to cost, I'm not convinced that this always works in favour of 8 bit processors. In terms of power, particularly, the low end ARM processors massively outperform an 8051 in terms of performance/microwatt. I hesitate to use Dhrystone as an example, as it is very obviously skewed towards 32-bit integer performance, but the ARM parts will give around 30-40x the DMIPS/mW figure of a "low-power" 8051. Every application I have looked at takes a lot less power on a 32-bit micro. That's nothing to do with clever power saving process technology, it's just because the raw architecture is so much more efficient.

Code density is the other thing that can skew the system cost in favour of a 32-bit processor. We are taking carefully written 8051 assembler + C and replacing it with C code running on an ARM and the memory footprint is typically much less on the ARM - a single 16 bit instruction operating on 32 bits of data can do a lot more work than two 8 bit instructions operating on 8 bits of data in a single accumulator. If the cost of your device is dominated by the on-board flash or SRAM, then an 8-bit processor may well work out more expensive just because it needs more of it, for non-trivial applications.

As engineers, we like to do things as efficiently as possible and using a 32-bit micro to watch a few inputs and toggle some output bits and drive some LEDs seems disproportionate. Moving from 8051 to Cortex-M3 sometimes seems like going from a horse and cart to a sports car (when a model T would've done the job just as well.) One of the consequences of this is that there tends to be major feature creep when migrating - it seems silly not do _something_ with all that spare performance?

Criminal Coding

Pierre-Jean Turpeau commented on my statement: “Code testing is a notoriously ineffective way to uncover bugs. How many of us couple effective tests with code coverage checks or inspections?"

In aerospace, the DO-178B (guideline for certifiable avionics software developments) ask for requirement testing at the component level. Of course, you need functional requirements to limit ineffective structural testing.

The DO-178B also ask for 100% code coverage and MCDC (multiple condition decision coverage) for the most critical parts of the software.

Functional requirements at the component level (file, function, class or package depending on your development strategy) are sometimes hard to write and require a good knowledge of how differentiate and separate the "what" (responsibilities of the component) and the "how" (the way it implements the responsibilities).

Now based on that, in a requirement testing process with 100% MCDC code coverage every branch shall be justified by a functional need.

This leads to two facts :
- You need highly tailored coding standards,
- You shouldn't write complex code unless you're happy with cost increase.

This means also that the simpler your code is, the safer you'll get...

And it's something that is carefully checked by authorities during the certification process.

Chuck Royalty weighed in, too: I think the root of the issue is that your last sentence is not true: Software is the only industry left on the planet that gets away with selling known defective products.

Most software is not sold; it's leased (licensed). The terms of the lease disclaim responsibility on the part of the author for almost everything, and that's legally ok because lease contracts are not subject to product fitness standards. On the other hand, if my car's brakes fail and it can be shown that they failed I (or my survivors) don't care if it was software or a jammed piston. If a product that's sold to me doesn't work (for appropriate values of "work") the manufacturer is liable; even more so if it causes injury.

So what about the software? My perception, and it's only that, is that the difference between the terms and conditions offered for the software to the product manufacturer and the liability of the manufacturer for product level effects of malfunction of that software, is borne by the product manufacturer.

Seems like manufacturers that sell computer-based products might care a whole bunch about that (and in fact I think they do, by and large).

Christoph Schmidt submitted: In your last muse you wrote about criminal coding by not implementing checks for buffer overflows. In my daily work I have the impression that approximately 80% of the severe bugs are caused by either buffer overflows or type casts.

One thing I don't understand about the embedded software world is why we still use programming languages that allow implementing erroneous code that easily. Using languages like Ada makes it much more difficult or nearly impossible to write down such crap. Even the best programmers forget from time to time to check for overflow or add a type cast which will turn into a very harmful construct as soon as one of the types is slightly changed. So why are we still using these languages?

First of all, it's quite easy to find embedded engineers familiar with C, other programming languages are by far less known. And tight project schedules normally don't allow learning a new language.

Many embedded engineers only know C and maybe assembler, they don't know about the advantages of other languages. They think the shortcomings of C are God-given and the way the world should be. I once had a discussion with one of my former colleagues about programming style, especially concerning the need for security checks and comments. His point of view was just "We are firmware engineers. We know what we are doing!" (cited) This mindset, that programming has to be a difficult low-level task and only a bright mind must be able to understand the code makes it nearly impossible to switch to a clearer and easier to read language.

Most of the embedded code around there is most likely written in C, making it more difficult to reuse or extend it using another language. But at least with gnat/gcc, Ada code can be linked against C code, so it's not impossible to switch to another programming language step by step without replacing the complete code base at once.

In many cases management does not know about other programming languages or they find it too dangerous to switch the language: You don't get fired because you use C. But if you introduced another language and the project failed, you are hard pressed to deliver a good explanation for the change.

Finally, a very common argument against higher-level languages are memory footprint and execution time. I haven't tried it yet myself, but I would bet that C code containing the same number of security checks as Ada would have nearly the same footprint and speed.

[Sorry for always using Ada as comparison. It's the only language suitable for embedded programming I know apart from C ...]

In the last clause you wrote that the software industry is the only industry which can afford to sell buggy products. Here I disagree. I once worked with an ASIC containing that many bugs that a colleague of mine replaced the word "BSP" with "BAL (Bug Adaptation Layer)" in our diagrams. I can't remember how many bugs this damn thing had but it was a lot.

Reuse

Bill Clare wrote: I wanted to weigh in on Roland Bennett's reuse comments. I've done a number of Operating System abstraction layers in my career (or, more appropriately, a single one that I've ported a bunch of times; pSOS/C to vxWorks/C to Windows/C to vxWorks/C++ to Nucleus C to Green Hills Integrity C++...). I've also seen articles on it and the argument always offered is the one that Roland illustrates below; portability. In and of itself it's reason enough to do the abstraction layer, but what I've seen is that there is another benefit that is perhaps just as important and certainly is not as well documented. With an abstraction layer, you can enforce desired behavior, best practices, design standards, etc. A couple of examples from my past:

- ATI's Nucleus (at least the version we used...) required the application space to allocate the memory used for the OS resource control blocks(task, semaphore, memory segment, etc., whether with a local variable, off the heap, using a class member, etc.). My abstraction layer hid these details from the end user, ensuring consistent management of memory used for OS resources.

- vxWorks' mutual exclusion primitive provides a priority inversion option that we hard-coded to "on" in our abstraction layer; we simply didn't expose this option in our abstraction layer's API.

Chris Eddy wanted to add one thought and reinforce another: The newsletter points out many specific techniques to make code more robust, but my biggest experience is that if the code is well planned out, in great detail, it accomplishes these things:

- The client has to specifically identify what they want and how they want it to look/work. On paper.

- I have a road map up front that lets me scale things to that I am not trying to cross rivers on alligators later.

- I can keep the customer firmly attached to a document as the inevitable changes come along. All I have to do is double the space that is needed for their revision 1, as it inevitably grows by 60%.

I find that this keeps consistent code better than the specific methods.

And to reinforce Roland Bennett's points, I do love to abstract things as much as possible. A customer recently asked me to go from an 8 bit processor to a 16 bitter (they actually defined twice the features, I defined the part), and I managed to move the 8 bit code over to the 16 bit part in just a few days. I cannot say enough about abstraction. And #defines for any 'flexible constant', such as the definition of a millisecond, et cetera. Then when my new processor has a different interrupt rate, I just clean up the #defines, and I am off and compiling.

I even take it as far as abstracting with compilers in mind. My major accounts use Microchip parts, and my code is prepared for both Hi-Tech and Microchip compilers. It seems redundant, but it does two things for me:

- When one compiler shows a crack with a flaw, I can quickly bail instead of waiting for the 'fix'.

- I can make use of a specific compilers' libraries, when I find out later that they want to add 'USB' after the fact, and one compiler is better supported than the other.

And furthermore… I inherit a lot of code, as a consultant. I can actually identify their age/era by their code (yup, learned to program on a PDP-11 or some such) or (yup.. I, J, and K.. started out as a FORTRAN guy).

Early on the embedded processors were small and you had to pack bits in a byte or word. Now, with SOOO much space in there, it is actually safer from a byte sharing standpoint to spread out your flags over separate bytes. In the 16 bit parts, I actually waste words. I say this because I see share overlaps with interrupts far more often than I see 'out of RAM space'.

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.

Horizon Hobby, Inc. is seeking a Senior Engineer - Electrical for work in the Proprietary Product Development Department at our world headquarters in Champaign, Illinois. Full details are available online at http://www.horizonhobby.com/Horizon/Careers.aspx This is the ideal job for an RC hobbyist with experience in development of electronics used in the consumer electronics/hobby arenas.

Insight Technology Incorporated, headquartered in Londonderry, NH is actively searching for embedded software engineers. You will be part of a small engineering team in a challenging, fast-paced company where designs become production reality. Details can be found at www.insighttechnology.com

Joke for the Week

From Paul Bennett:

Lawrence Livermore Laboratories has discovered the heaviest element yet known to science.

The new element, Governmentium (Gv), has one neutron, 25 assistant neutrons, 88 deputy neutrons, and 198 assistant deputy neutrons, giving it an atomic mass of 312.

These 312 particles are held together by forces called morons, which are surrounded by vast quantities of lepton-like particles called peons.

Since Governmentium has no electrons, it is inert; however, it can be detected, because it impedes every reaction with which it comes into contact. A tiny amount of Governmentium can cause a reaction that would normally take less than a second, to take from 4 days to 4 years to complete.

Governmentium has a normal half-life of 2- 6 years. It does not decay, but instead undergoes a reorganization in which a portion of the assistant neutrons and deputy neutrons exchange places. In fact, Governmentium's mass will actually increase over time, since each reorganization will cause more morons to become neutrons, forming isodopes.

This characteristic of morons promotion leads some scientists to believe that Governmentium is formed whenever morons reach a critical concentration. This hypothetical quantity is referred to as critical morass.

When catalysed with money, Governmentium becomes Administratium, an element that radiates just as much energy as Governmentium since it has half as many peons but twice as many morons

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.