A Guide to Commenting

Jack G. Ganssle
jack@ganssle.com

The Ganssle Group
PO Box 38346
Baltimore, MD 21231
(410) 496-3647
fax (647) 439-1454

According to Henry Petroski (reference 1), the first known book about engineering is the 2000 year old work “De Architectura” by Marcus Vitruvius Pollio. It’s a fairly complete description of how these skilled artisans created their bridges and tunnels in ancient Rome.

One historian said of Vitruvius and his book: “He writes in atrocious Latin, but he knows his business”. Another wrote: “He has all the marks of one unused to composition, to whom writing is a painful task”.

How little things have changed! Even two millennia ago engineers wrote badly, yet were recognized as experts in their field. Perhaps even then these Romans were geeks. Were engineers from Athens Greek geeks?

Some developers care little about their poor writing skills, figuring they interact with machines, not people. And of course we developers just talk to other writing-challenged engineers anyway, right?

Wrong.

This is the communications age. The spoken and written word has never been more important. Consider how email has reinvigorated letter-writing…. yet years ago I remember hearing philologists moaning about the death of letters.

Old timers will remember how engineers could once function perfectly with no typing skills. That seems quaint today, when most of us live with a keyboard all but strapped to our hands. Just as old-fashioned is the idea of a secretary transcribing notes and the fixing spelling and grammar. Today it’s up to *us* to express ourselves clearly, with only the assistance of a spellchecker and an annoyingly-picky grammar engine.

I write a weekly column on embedded.com which generates quite a bit of feedback via email. The majority of these responses are quite well written, giving lie to the old generalization of engineers being compositionally challenged.. But some replies are rather appalling. Obviously non-English speakers struggle with our language’s idiosyncrasies. But all too many of these confusing ungrammatical missives come from Joe Smith in Anytown, USA.

Even if you’re stuck in a hermitically-sealed cubicle never interacting with people and just cranking code all day, I contend you still have a responsibility to communicate clearly and grammatically with others. Software is, after all, a mix of computerese (the C or C++ itself) and comments (in America, at least, an English-language description meant for humans, not the computer). If we write perfect C with illegible comments, we’re doing a lousy job.

© 2001 The Ganssle Group. This work may be used by individuals and companies, but all publication rights reserved.

I read a *lot* of code from a huge range of developers. Consistently well-done comments are rare. Sometimes I can see the enthusiasm of the team at the project's outset. The startup code is fantastic. Main()'s flow is clear and well documented. As the project wears on functions get added and coded with less and less care. Comments like

```
/* ???? */
```

or my favorite:

```
/* Is this right? */
```

start to show up. Commenting frequency declines; clarity gives way to short cryptic notes; capitalization descends into chaotic randomness. The initial project excitement, as shown in the careful crafting of early descriptive comments, yields to schedule panic as the developers all but abandon anything that's not executable.

Onerous and capricious schedules are a fact of life in this business. It's natural to chuck everything not immediately needed to make the product work. Few bosses grade on quality of the source code. Quality, when considered at all, is usually a back-end complaint about all the bugs that keep surfacing in the released product, or the ongoing discovery of defects that pushes the schedule back further and further.

Pride

We firmware folks know that quality starts at the front-end, in proper design and implementation, using reasonable processes. Quality also requires fine workmanship. Our profession parallels that of the trade crafts of centuries ago. The perfect joint in a chair may be almost invisible, but will last forever. A shoddy alternative could be just as hard to see, but is simply not acceptable. Professional pride mandates doing the right thing just because we know it's the best way to build the product.

Most of us create software in secret. I rarely see companies using code inspections, for example, which at the very least brings our flaws into the cold harsh light of day. Secrecy naturally breeds laziness. It takes a very strong person to consistently rise above the temptations of expediency to do things *right*, even when it's not clear that there will be a reward for working carefully.

Though we embedded people work at the border between hardware and software, where sometimes it's hard to say where one ends and the other starts, even hardware designers work in the spotlight. Their creations are subject to ongoing audits during manufacturing, test and repair. Technicians work with the schematics daily. Faults glare from the page for everyone to see. Sloppy work can't be hidden.

© 2001 The Ganssle Group. This work may be used by individuals and companies, but all publication rights reserved.

(Now, though, ASICs, programmable logic and high level synthesis can bury lots of evil in the confines of an inscrutable IC package. The hardware folks are inheriting all of the perils of software.)

I'm fascinated by eXtreme Programming, though shudder at some of the practices it espouses. Al of XP's ideas come from four "core values": communications, simplicity, feedback and courage. No other methodology that I'm aware of derives from *values*. In America we talk a lot about values, sometimes so much so that the meaning gets lost in the rhetoric. Yet values of all sorts are the basis of good behavior. I think the XP folks got it right by deriving the process from values rather than from a collection of good ideas. However, I'd add a fifth to their list: Pride of Workmanship.

In my experience software created without pride is awful. Shortcuts abound. The limited docs never mirror current reality. Error conditions and exceptions are poorly thought-out. For example, Microsoft's various products have garnered a reputation for their susceptibility to buffer overflow attacks. Unix, too, has long suffered the same flaws. Recent posts on the Risks forum http://catless.ncl.ac.uk/Risks/21.84.html and http://catless.ncl.ac.uk/Risks/21.85.html suggest that the C language is the source of the problem. Programs written in C usually have no intrinsic array bounds checking; worse, the dynamic nature of pointers makes automatic run time checks that much more problematic.

I disagree. C is nothing more than a tool, one that should come with an "adults only" warning. Those who use it carelessly are at fault, not the language itself. Index into a data structure without adding the requisite overflow checks and you're playing with dynamite. While smoking. In a puddle of gasoline.

Every programmer knows he or she should run simple sanity checks on all data from untrusted sources. Not doing so is laziness, a lack of Pride in Workmanship. Careful craftsmen spend a few seconds adding these checks to save months of debugging or millions in product recalls.

Commenting Suggestions

My standard for commenting is that someone versed in the functionality of the product – but not the software – should be able to follow the program flow by reading the comments without reference to the code itself. Code implements an algorithm; the comments communicate the code's operation to yourself and others. Maybe even to a future version of yourself during maintenance years from now.

Write every bit of the documentation (in the USA at least) in English. Noun, verb. Use active voice. Be concise; don't write the Great American Novel. Be explicit and complete; assume your reader has not the slightest insight into the solution of the problem. In most cases I prefer to

© 2001 The Ganssle Group. This work may be used by individuals and companies, but all publication rights reserved.

incorporate an algorithm description in a function's header, even for well-known approaches like Newton's Method. A description that uses your variable names makes a lot more sense than "see any calculus book for a description." And let's face it: once carefully thought out in the comments it's almost trivial to implement the code.

Capitalize per standard English procedures. IT HASN'T MADE SENSE TO WRITE ENTIRELY IN UPPER CASE SINCE THE TELETYPE DISAPPEARED 25 YEARS AGO. the common c practice of never using capital letters is also obsolete. Worst aRe the DevElopeRs wHo usE rAndOm caSe changeS. Sounds silly, perhaps, but I see a lot of this. And spel al of the wrds gud.

Avoid long paragraphs. Use simple sentences. "Start_motor actuates the induction relay after a three second pause" beats "this function, when called, will start it all off and flip on the external controller but not until a time defined in HEADER.H goes by."

Begin every module and function with a header in a standard format. The format may vary a lot between organizations, but should be consistent within a team. Every module (source file) must start off with a general description of what's in the file, the company name, a copyright message if appropriate, and dates. Start every function with a header that describes what the routine does and how, goes-intas and goes-outas (i.e., parameters), the author's name, date, version, a record of changes with dates and the name of the programmer who made the change.

C lends itself to the use of asterisks to delimit comments, which is fine. I see a lot of this:

```
/**************
*  comment       *
**************/
```

which is a lousy practice. If your comments end with an asterisk as shown, every edit requires fixing the position of the trailing asterisk. Leave it off, as follows:

```
/**************
*  comment
**************/
```

Most modern C compilers accept C++'s double slash comment delimiters, which is more convenient than the /* */ C requires. Start each comment line with the double slash so the difference between comments and code is crystal clear.

Some folks rely on a fancy editor to clean up comment formatting or add trailing asterisks. Don't. Editors are like religion. Everyone has their own preference, each of which is configured differently. Someday compilers will accept source files created with a word processor which

© 2001 The Ganssle Group. This work may be used by individuals and companies, but all publication rights reserved.

will let us define editing styles for different parts of the program. Till then dumb ASCII text formatted with spaces (not tabs) is all we can count on to be portable and reliable.

Enter comments in C at block resolution and when necessary to clarify a line. Don't feel compelled to comment each line. It is much more natural to comment groups of lines which work together to perform a macro function.

Explain the meaning and function of every variable declaration. Long variable names are merely an *aid* to understanding; accompany the descriptive name with a deep, meaningful, prose description.

One of the perils of good comments – which is frequently used as an excuse for sloppy work – is that over time the comments no longer reflect the truth of the code. Comment drift is intolerable. Pride in Workmanship means we change the docs as we change the code. The two things happen in parallel. Never defer fixing comments till later, as it just won't happen. Better: edit the descriptions first, then fix the code.

One side effect of our industry's inglorious 50 year history of comment drift is that people no longer trust comments. Such lack of confidence leads to even sloppier work. It's hard to thwart this descent into commenting chaos. Wise developers edit the header to reflect the update for each patch, but even better add a note that says "comments updated, too" to build trust in the docs.

If you use code inspections (and please do – they are the cheapest known way to get rid of bugs. See http://ganssle.com/inspections.htm for a description) review the comments as well as the code. Both are equally important.

Finally, consider changing the way you write a function. I have learned to write all of the comments first, including the header and those buried in the code. Then it's simple, even trivial, to fill in the C or C++. Any idiot can write software following a decent design; inventing the design, reflected in well-written comments, is the really creative part of our jobs.

Reference 1: The Pencil : A History of Design and Circumstance by Henry Petroski (December 1992) Knopf; ISBN: 0679734155

© 2001 The Ganssle Group. This work may be used by individuals and companies, but all publication rights reserved.

Better Firmware... *Faster!*

A One-Day Seminar

Presented at

Your Company

Does your schedule prevent you from traveling?

This doesn't mean you have to pass this great opportunity by.

Presented by Jack Ganssle, technical editor of *Embedded Systems Programming Magazine*, **author** of *The Art of Developing Embedded Systems, The Art of Programming Embedded Systems,* and *The Embedded Systems Dictionary*

More information at www.ganssle.com

The Ganssle Group
PO Box 38346
Baltimore, MD 21231
(410) 496-3647
fax: (647) 439-1454
info@ganssle.com
www.ganssle.com

For Engineers and Programmers

This seminar will teach you new ways to build higher quality products in half the time.

80% of all embedded systems are delivered late…
Sure, you can put in more hours. Be a hero. But *working harder is not a sustainable way to meet schedules.* We'll show you how to plug productivity leaks. How to manage creeping featurism. And ways to balance the conflicting forces of schedules, quality and functionality.

… yet it's not hard to double development productivity
Firmware is the most expensive thing in the universe, yet we do little to control its costs. Most teams deliver late, take the heat for missing the deadline, and start the next project having learned nothing from the last. Strangely, *experience* is not correlated with *fast*. But *knowledge* is, and we'll give you the information you need to build code more efficiently, gleaned from hundreds of embedded projects around the world.

Bugs are the #1 cause of late projects…
New code generally has *50 to 100 bugs* per thousand lines. Traditional debugging is the *slowest* way to find bugs. We'll teach you better techniques proven to be up to 20 times more efficient. And show simple tools that find the nightmarish real-time problems unique to embedded systems.

… and poor design creates the bugs that slip through testing
Testing is critical, but it's a poor substitute for well-designed code. Deming taught us that you cannot create quality via testing. We'll show you how to create *great designs* that intrinsically yield better code in less time.

Learn From The Industry's Guru

Spend a day with Jack Ganssle, well-known author of the most popular books on embedded systems, technical editor and columnist for Embedded Systems Programming, and designer of over 100 embedded products. You'll learn new ways to produce projects *fast* without sacrificing quality. This seminar is the only non-vendor training event that shows you *practical* solutions that you can implement *immediately*. We'll cover technical issues – like how to write embedded drivers and isolate performance problems – as well as practical process ideas, including how to manage your people and projects.

Seminar Leader

Jack Ganssle has written over 300 articles in Embedded Systems Programming, EDN, and other magazines. His three books, The Art of Programming Embedded Systems, The Art of Developing Embedded Systems, and his most recent, The Embedded Systems Dictionary are the industry's standard reference works

Jack lectures internationally at conferences and to businesses, and was this year's keynote speaker at the Embedded Systems Conference. He founded three companies, including one of the largest embedded tool providers. His extensive product development experience forged his unique approach to building better firmware faster.

Jack has helped over 600 companies and thousands of developers improve their firmware and consistently deliver better products on-time and on-budget.

Course Outline

Languages

- C, C++ or Java?
- Code reuse – a myth? How can you benefit?
- Stacks and heaps – deadly resources you *can* control.

Structuring Embedded Systems

- *Manage* features… or miss the schedule!
- Do commercial RTOSes make sense?
- Five design schemes for faster development.

Overcoming Deadline Madness

- Negotiate realistic deadlines… or deliver late.
- Scheduling – the science versus the art.
- Overcoming the biggest productivity busters.

Stamp Out Bugs!

- Unhappy truths of ICEs, BDMs, and debuggers.
- *Managing* bugs to get good code fast.
- *Quick* code inspections that keep the schedule on-track.
- Cool ways to find hardware/software glitches.

Managing Real-Time Code

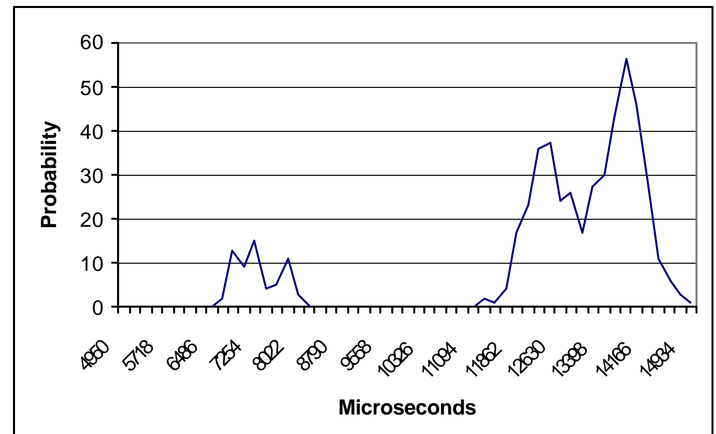
- Design *predictable* real-time code.
- Preventing system performance debacles.
- Troubleshooting and eliminating *erratic crashes.*
- Build better interrupt handlers.

Interfacing to Hardware

- Understanding high-speed signal problems.
- Building peripheral drivers faster.
- Cheap – and expensive – ways to probe SMT parts.

How to Learn from Failures… and Successes

- Embedded disasters, and *what we can learn*.
- Using postmortems to accelerate the product delivery.
- Seven step plan to firmware success.



Do those C/C++ runtime routines execute in a usec or a week? This trig function is all over the map, from 6 to 15 msec. You'll learn to rewrite real-time code proactively, anticipation timing issues before debugging.

Why Take This Course?

Frustrated with schedule slippages? Bugs driving you batty? Product quality sub-par? **Can you afford *not* to take this class?**

We'll teach you how to get your products to market faster with fewer defects. Our recommendations are *practical*, *useful today*, and *tightly focused* on embedded system development. Don't expect to hear another clever but ultimately discarded software methodology. You'll also take home a 150-page handbook with algorithms, ideas and solutions to common embedded problems.

If you can't take the time to travel, we can present this seminar at your facility. We will train all of your developers and focus on the challenges unique to your products and team.

Here is what some of our attendees have said:

Thanks for the terrific seminar here at ALSTROM yesterday!
It got rave reviews from a pretty tough crowd.
Cheryl Saks, ALSTROM

Thanks for a valuable, pragmatic, and informative lesson in embedded systems design.
All the attendees thought it was well worth their time.
Craig DeFilippo, Pitney Bowes

I just wanted to thank you again for the great class last week. With no exceptions, all of the feedback from the participants was extremely positive. We look forward to incorporating many of the suggestions and observations into making our work here more efficient and higher quality.
Carol Bateman, INDesign LLC

Here are just a few of the companies where Jack has presented this seminar:
Sony-Ericsson, Northup Grumman, Dell, Western Digital, Bayer, Seagate, Whirlpool, Cutler Hammer, Symbol, Visteon, Honeywell, Kodak and Western Digital.

Did you know that…

… doubling the size of the code results in much more than twice the work? In this seminar you'll learn ways unique to embedded systems to partition your firmware to keep schedules from skyrocketing out of control.

… you can reduce bugs by an order of magnitude before starting debugging? Most firmware starts off with a 5-10% error rate – 500 or more bugs in a little 10k LOC program. Imagine the impact finding all those has on the schedule! Learn simple solutions that don't require revolutionizing the engineering department.

… you can create a predictable real-time design? This class will show you how to measure the system's performance, manage reentrancy, and implement ISRs with the least amount of pain. You'll even study real timing data for common C constructs on various CPUs.

… a 20% reduction in processor loading slashes development time? Learn to keep loading low while simplifying overall system design.

… reuse is usually a waste of time? Most companies fail miserably at it. Though promoted as the solution to the software crisis, real reuse is much tougher than advertised. You'll learn the ingredients of successful reuse.

What are you doing to upgrade your skills? What are you doing to help your engineers succeed? Do you consistently produce quality firmware on schedule? *If not . . . what are you doing about it?*

Contact us for info on how we can bring this seminar to your company.
e-mail: info@ganssle.com or call us at 410-496-3647.