A Guide to Code Inspections

Jack G. Ganssle
jack@ganssle.com

The Ganssle Group
PO Box 38346
Baltimore, MD 21231
(410) 496-3647
fax (410) 675-2245

There *is* a silver bullet that can drastically improve the rate you develop code while also reducing delivered bugs.

Though this bit of magic can reduce debugging time by an easy factor of 10 or more, despite the fact that it's a technique well known since 1976, and even though neither tools nor expensive new resources are needed, few embedded folks use it.

Formal code inspections are probably the most important tool you can use to get your code out faster with fewer bugs. The inspection plays on the well-known fact that "two heads are better than one". The goal is to identify and remove bugs *before* testing the code.

Effectiveness

Those that are aware of the method often reject it because of the assumed "hassle factor". Usually few developers are aware of the benefits that have been so carefully quantified over time. Let's look at some of the data.

The very best of inspection practices yield stunning results. For example, IBM manages to remove 82% of all defects *before* testing even starts!

One study showed that, as a rule of thumb, each defect identified during inspection saves around 9 hours of time downstream.

AT&T found inspections led to 14% increase in productivity and tenfold increase in quality.

HP found 80% of the errors detected during inspections were unlikely to be caught by testing.

HP, Shell Research, Bell Northern, and AT&T all found inspections 20 to 30 times more efficient at testing in detecting errors.

IBM found inspections gave a 23% increase in productivity and a 38% reduction in bugs detected after unit test.

So, though the inspection may cost you 20% extra time during coding, debug times can shrink by an order of magnitude or more. The reduced number of bugs in the final product means you'll spend less time in the mind-numbing weariness of maintenance as well.

The Inspection Team

The best inspections come about from properly organized teams. Keep management off the team! Experience indicates that when a manager is involved usually only the most superficial bugs are caught, since no one wishes to show the author to be the cause of major program defects.

Four formal roles exist: the Moderator, Reader, Recorder, and Author.

The Moderator, who must be very competent technically, leads the inspection process. He or she paces the meeting, coaches other team members, deals with scheduling a meeting place and disseminating materials before the meeting, and follows up on rework (if any).

The Reader takes the team through the code by paraphrasing its operation. Never let the Author take this role, since he may read what he meant instead of what he implemented.

A Recorder notes each error on a standard form. This frees the other team members to focus on thinking deeply about the code.

The Author's role is to understand the errors found and to illuminate unclear areas. As code inspections are never confrontational, the Author should never be in a position of defending the code.

An additional role is that of Trainee. No one seems to have a clear idea how to create embedded developers. One technique is to include new folks (only one or two per team) into the code inspection. The Trainee(s) then get a deep look inside of the company's code, and an understanding of how the code operates.

It's tempting to reduce the team size by sharing roles. Bear in mind that Bull HN found four person inspection teams are twice as efficient and twice as effective as three person teams. A code inspection with three people (perhaps using the Author as the Recorder) surely beats none at all, but do try to fill each role separately.

The Process

Code inspections are a *process* consisting of several steps; all are required for optimal results. The steps are:

Planning - When the code compiles cleanly (no errors or warning messages), and after it passes through Lint (if used) the Author submits listings to the Moderator, who forms an inspection team. The moderator distributes listings to each team member, as well as other related documents such as design requirements and documentation. The bulk of the Planning

process is done by the Moderator, who can use email to coordinate with team members. An effective Moderator respects the time constraints of his colleagues and avoids interrupting them.

Overview - This optional step is a meeting for cases where the inspection team members are not familiar with the development project. The Author provides enough background to team members to facilitate their understanding of the code.

Preparation - Inspectors individually examine the code and related materials. They use a checklist to ensure they check all potential problem areas. Each inspector marks up his or her copy of the code listing with suspected problem areas.

Inspection Meeting - The entire team meets to review the code. The Moderator runs the meeting tightly. The only subject for discussion is the code under review; any other subject is simply not appropriate and not allowed.

The person designated as Reader presents the code by paraphrasing the meaning of small sections of code in a context higher than that of the code itself. In other words, the Reader is translating short code snippets from computer-lingo to English to ensure the code's implementation has the correct meaning.

The Reader continuously decides how many lines of code to paraphrase, picking a number that allows reasonable extraction of meaning. Typically he's paraphrasing 2-3 lines at a time. He paraphrases every decision point, every branch, case, etc. One study concluded that only 50% of the code gets executed during typical tests, so be sure the inspection looks at *everything*.

Use a checklist to be sure you're looking at all important items. See the "Code Inspection Checklist" for details.

Record all errors and classify them as Major or Minor. A Major bug is one that if not removed could result in a problem that the customer will see. Minor bugs are those that include spelling errors, non-compliance with the firmware standards, and poor workmanship that does not lead to a major error.

Why the classification? Because when the pressure is on, when the deadline looms near, management will demand that you drop inspections as they don't seem like "real work." A list of classified bugs gives you the ammunition needed to make it clear that dropping inspections will yield more errors and slower delivery.

Two forms get filled out. The "Code Inspection Checklist" is a summary of the number of errors of each type that's found. It's used for understanding how effective the inspection process is.. The "Inspection Error List" is the details of each error requiring rework.

The code itself is the only thing under review; the author may not be criticized. One effective way of defusing the tension in starting up new inspection processes (before the team members are truly comfortable with it) is to have the Author supply a pizza for the meeting. Then he seems like the good guy.

At this meeting no attempt is made to rework the code, or to come up with alternative approaches. Find errors and log them; let the Author deal with implementing solutions. The Moderator must keep the meeting fast-paced and efficient. In fact, a reasonable review rate is between 150 and 200 non-blank lines per hour.

Note that comment lines require as much review as code lines. Misspellings, lousy grammar, and poor communication of ideas are as deadly in comments as outright bugs in code. Firmware must do two things to be acceptable: it must work, and it must communicate its meaning to a future version of yourself - and to others. The comments are a critical part of this and deserve as much attention as the code itself.

It's worthwhile to compare the size of the code to the estimate originally produced (if any!) when the project was scheduled. If it varies significantly from the estimate figure out why, so you can learn from your estimation process.

Limit inspection meetings to a maximum of two hours. At the conclusion of the review of each function decide whether the code should be accepted as is or sent back for rework.

Rework - The Author makes all suggested corrections, gets a clean compile (and Lint if used) and sends it back to the Moderator.

Follow-up - The Moderator checks the reworked code. If the Moderator is satisfied the inspection is formally complete and the code may be tested.

Other Points

One hidden benefit of code inspections is their intrinsic advertising value. We talk about software reuse, while all too often failing spectacularly at it. Reuse is certainly tough, requiring a lot of discipline and work. One reason it fails, though, is simply because people are not aware of the code. If you don't know there's a function on the shelf, ready to rock 'n roll, then there's no chance you'll reuse it. The inspection makes more people aware of what code exists.

The literature is full of the pros and cons of inspecting code before you get a clean compile. My feeling is that the compiler is nothing more than a tool, one that very cheaply and quickly picks up the stupid silly errors we all make. Compile first, and let the tool rather than expensive people pick up the simple mistakes.

Along those same lines, I also believe that the only good compile is a clean compile. No error messages. No warning messages. Warnings are deadly when some other programmer, maybe years from now, tries to change a line. When presented with a screenful of warnings he'll have no idea if these are normal or a symptom of a problem.

Conclusion

Inspections break the dysfunctional code-compile-debug cycle. We know firmware is hideously complex and awfully prone to failure. It's crystal clear from data, both quantitative and anecdotal, that code inspections are the cheapest and most effective bug beaters in the known universe. Yet few organizations, especially smaller ones, use them on their firmware.

Inspect *all* of your code. Make this a habit. Resist the temptation to abandon inspections when the pressure heats up. Being a software professional means we do the right things, all of the time. The alternative is to be a hacker - cranking the code out at will with no formal discipline.

Inspection shouldn't be limited to code; all specification and design documents benefit from a similar process.

For those interested in more information, check out the following two books (both very highly recommended). Both are available from the Computer Literacy Bookstore and amazon.com:

Software Inspection, Tom Gilb and Dorothy Graham, 1993, TJ Press (London). ISBN 0-201-63181-4.

Software Inspection - An Industry Best Practice, David Wheeler, Bill Brykczynski and Reginald Meeson, 1996 by IEEE Computer Society Press (CA), ISBN 0-8186-7340-0.

Code Inspection Checklist

Project: _____
Author: _____
Function Name: _____
Date: _____

Number of errors		*Error Type*
Major	Minor	
		Code does not meet firmware standards
		Function size and complexity unreasonable
		Unclear expression of ideas in the code
		Poor encapsulation
		Function prototypes not correctly used
		Data types do not match
		Uninitialized variables at start of function
		Uninitialized variables going into loops
		Poor logic - won't function as needed
		Poor commenting
		Error condition not caught (e.g., return codes from malloc())?
		Switch statement without a default case (if only a subset of the possible conditions used)?
		Incorrect syntax - such as proper use of ==, =, &&, &, etc.
		Non reentrant code in dangerous places
		Slow code in an area where speed is important
		Other
		Other

A Major bug is one that if not removed could result in a problem that the customer will see. Minor bugs are those that include spelling errors, non-compliance with the firmware standards, and poor workmanship that does not lead to a major error.

Inspection Error List

Project: _____
Author: _____
Function Name: _____
Date: _____
Rework required? _____

Location	*Error Description*	*Major*	*Minor*

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

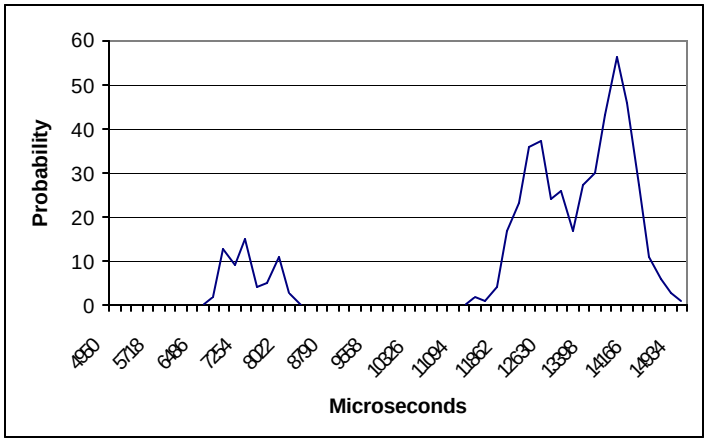
- Design *predictable* real-time code.
- Preventing system performance debacles.
- Troubleshooting and eliminating *erratic crashes.*
- Build better interrupt handlers.

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.