Measuring Bugs

subscribe to the Embedded Muse.

Measuring Bugs

It seemed a simple project, really. The 32k ROM constraint set a reasonable limit on firmware size and complexity. Bob, the solo developer, had built several of these controllers for his company; one more wouldn't tax his skills.

His off-the-cuff estimate of 5k lines of C started him down the thorny path of complacency. Turns out the guess wasn't that far off, but the minimal size lured him into hasty design sans reviews. Within a week of starting the project he was cranking code. m mso-bidi-font-family:"Times New Roman"">ain() was a simple loop; the ISRs trivial, really. Functions sprouted like weeds, none particularly hard but each interacting in increasingly complex ways.

Well ahead of schedule he started debugging! and the project became mired in problems. Weeks turned into months, months into quarters, and each day more legions of elusive bugs appeared. It seemed fixing one created or unmasked three others. The due date came and went, pestering QA people never left Bob alone, and the boss's demands grew ever more shrill. The project was eventually cancelled.

Sound familiar? How about this one:

The team lead was a graduate of many a development effort, and had honed his skills by constantly searching each completed project for lessons learned. A thick veneer of *process* mediated every phase of design, coding and implementation. Every intermediate due date was met on time and without fuss. Design completed, coding mostly done, developers fired up their debuggers and quickly tested most of the application-level code.

But this was an embedded system, and the lead's experience largely came from creating business packages. Rumbles of discontent from the application guys grew louder as their debugging efforts became stymied by the lack of low-level drivers.

The poor sods interfacing to the complex ASIC chipset seemed unable to produce anything that worked. Sure, they had tons of code! all of which behaved in unpredictable and buggy ways. Exasperated, unable to understand the peculiar environment of firmware developers, he demanded explanations. "The docs are wrong or non-existent; we don't know how these peripherals really work; the EEs keep inverting and transposing register bit assignments, and, well, at some point we had to make some assumptions about how things were supposed to work. These seem to be consistently wrong."

The original driver designs so cleanly transposed to C++ code were then changed, fixed, modified, and hacked to meet the seemingly inscrutable requirements of the ASIC, till the code was an ultra-tuned fragile assemblage no one understood. Every tiny apparently-benign edit led to the collapse of other, unrelated, functions. Schedules started slipping, the drivers never improved to any significant degree, and eventually the over-budget project was cancelled.

Both stories are true, and show that despite the best and worst of engineering practices an avalanche of bugs destroyed the projects.

Bugs are inevitable. Lousy engineering produces them but great engineering is no sure cure. The very best of processes will surely reduce their incidence, but cannot eliminate them! at least in the commercial world where time to market and costs are important concerns.

This suggests that wise developers will *manage* bugs, as aggressively as the legal department manages lawsuits. No one wants either suits or bugs, but since both are a fact of life it's critical we engage them up front, plan for them, and efficiently deal with them.

Last month I talked about making and using measurements in firmware development. It usually makes sense to look at how metrics can help us when we find consistent and expensive problems.

As I mentioned, managing by measurement is a Bad Thing in a creative environment like development. But managing without measurement is equally dysfunctional. The trick is to find a mix of reasonable metrics that yield important data easily.

To recapitulate my main message, never take data unless you're prepared to go through all four steps of the measurement process. Collect the data. Make sense of it - raw data is usually pretty raw. Present the data, to your boss, your colleagues, or perhaps even to just yourself. And, most importantly, act on the data. The only reason to collect software metrics is to enable change.

Stories and Stats

Over the years I've worked with an awful lot of engineering groups, and have observed that most devote somewhere around half of the schedule to debugging. Few actually plan for this much time, of course. Most usually have only a vague awareness that developers spend an awful lot of time in front of their debuggers.

It's astonishing how many development teams never even plan for bugs; debugging is almost an afterthought, something that despite repeated experience folks figure won't be much of an issue. They're wrong.

15 years in the In-Circuit Emulator business showed me a lot about how teams go about bug management. I observed, for instance, that 70% of our ICEs were shipped FedEx overnight. Why is nearly everyone operating in panic mode?

In most cases developers ran into real-time problems whose difficulty exceeded the capability of the tools. When planning the project their egos subliminally suggested that their work would be nearly-perfect. Reality hits, the ROM monitor just isn't up to tracking interrupts, tasks, DMA activity, interprocessor comm, or any of the other complexities that infest real systems, so a frantic search for more powerful tools ensues.

Take it from one who has watched this behavior hundreds of times. I'm long out of the tool business, have no interests in it anymore, so have no ax to grind. But good tools are cheap. Their price is usually irrelevant. Spend $10k or more on the right debugger and you'll save much more than that over the life of the project. Plan to buy the tools, early, rather than reacting to the problems that always surface. You'll save money.

A lot of developers maximized their costs by panicked rentals. Original contracts only very rarely exceeded three months; single-month rentals were quite common. Yet, practically without exception, customers extended the contracts, a month at a time, at the exorbitant single-month rate. By the time a year went by some chose to convert the rental to a purchase; many others continued the monthly rental extensions for two and three years.

The continuing optimism despite the evidence of these developers always astonished me and delighted our third party leasing company.

Monthly rentals typically morph to much longer terms. Initial efforts to save money end up costing dearly.

Tool vendors don't care if customers purchase, rent or lease their products. All have learned that "short term rental" almost always translates into "long term revenue stream." We warned our customers that they were usually better off just buying outright, but found resistance due to two reasons.

The first is the "just one more bug" syndrome. The product almost works. One intractable defect holds up shipment; when fixed, we're done. Why not spend $500 to rent a tool for one month?

After watching too many developers fall prey to this condition I've learned it's a sign of trouble. "Just one more bug and we're done" really means that the team is reacting to defects, desperately swatting them down as they appear. No one knows the real status of the product, but hopeful pronouncements to the boss keeps cancellation or layoffs at bay. For a while.

Monthly rentals were also more attractive than purchasing because bosses just don't get the debugging picture. They sense that bugs are a symptom of lousy engineers. "You want me to spend $10k on a tool that finds your mistakes?" How can one argue with that statement without appearing incompetent? No logical argument will overcome the boss's disgust. Hardware developers need several PCB spins. Doctors spend $100k a year on malpractice insurance. Airplanes are outfitted with black boxes in case of crashes. Most professions account for and expect mistakes. For some reason in software engineering this is not true. Yet our business is one of the most mistake-prone on the planet. One incorrect bit out of a hundred million can kill people. No one can honestly aspire to that level of perfection.

Metrics

If my observation that many of us spend half the project in debug is even close to correct, then optimizing debugging is the best way to accelerate schedules. As Michael Barr noted in his May editorial, we should all

commit to code inspections to find bugs cheaply. Inspections find bugs 20 times cheaper than conventional debugging. Don't believe me? Discount that by an order of magnitude and inspections are still twice as effective as debugging.

Beyond inspections, it pays to have a quantitative idea of where the bugs are. Did you know that the average software project - without inspections - runs about a 5% error rate (reference 1)? After getting a clean compile figure on one bug per twenty lines of code. Expect 500 bugs in a little 10K line project. No wonder half the schedule is eaten by debugging.

There are extremes: the Space Shuttle project runs one bug per 400,000 lines of code! at a cost of $1000/line (reference 2). That's taxpayer money, of course - there's plenty more where that came from. I've also seen too many developers with a 20% error rate. When someone says they can crank 200 lines of C per day, figure on massive back-end schedule slippages.

Count bugs. It's a sure way to improve the quality of the code. The process of taking such measurements leads to fewer bugs, and thus faster deliveries.

Barry Boehm showed (reference 3) that 80% of the bugs are in 20% of the modules. That confirms a feeling that many of us have. We know on most projects there are a few functions that are consistently infuriating. Change anything - even a comment - and the silly thing breaks. No one, not even the original designer, really knows how it works anymore. We try and beat the function into submission by constant tweaking but it's an exercise in futility.

Boehm showed that development costs for these few nuisance functions average 4 times any other function. So it's pretty obvious that if we can identify the problem modules quantitatively and early we can take corrective action cheaply. In other words, count bugs per function and be ready to toss out any chunk of code that stands out from the rest. Boehm's 4x factor means it's cheaper to toss it early than to beat it into submission.

Suppose Joe Coder is assigned function X. He hunches over the computer in his miniscule cubicle, trying to ignore the PA system and phone conversations going on around him, and creates the code. It gets inspected and only then does Joe start testing.

What Joe does during debugging is between him, the computer, and God. No one else needs to know the nature of his development sins. Nor is it practical to expect Joe to log errors. Few of us can live up to such a high standard of honesty.

But at some point he releases the function to the rest of the team, with his "tested" imprimatur stamped upon it. Perhaps one should say he *inflicts* it on the team, as every problem that surfaces now impacts the whole project.

This is the time to count and track bugs. As soon as more than a few appear it's time to look closely at the code and make hard decisions. Is the function turning into a nightmare? A constant source of trouble? If the bug rate is significantly higher than any other function, toss the code, start over, and get it right. Everyone is entitled to screw up occasionally, but we do know that the second time we'll get it right.

Don't just count the bugs. Log the nature of the problem. Go over the list from time to time looking for patterns, for they are sure to appear. Do you regularly seem to mix up "=" and "=="? Once the pattern is clear you're less likely to make the mistake in the first place.

Microsoft realized they had a recurrent pattern of buffer overflow bugs. Outlook and other programs were infested with unchecked input buffers, creating uncounted opportunities for hackers to create destructive virii. Understanding this problem, they reviewed all the code and checked each buffer use. All new code was similarly inspected. As a result the Windows environment became nearly virus-proof.

That story, of course, is a fantasy. Yet the pattern of problems is so apparent to their user community! The benefits of such an effort are obvious, the lack of such work almost criminal.

Conclusion

I collect embedded disaster stories as they are all so instructive. One that struck me was the 1994 crash of the Chinook helicopter. As is true in most aviation accidents many factors contributed to the crash. However, the engine control software was at least partially at fault.

After the accident a review board attempted to study the code (reference 3), giving up after reading less than 20% due to the vast number of issues found. Some 56 category 1 errors (outright coding errors or spec non-compliance) and over 400 other issues surfaced.

To me this is a prime example of dysfunctional bug measurement. Though probably necessary to assign blame and ultimately correct the problems, ex post facto software analysis always results from not taking the data up front, during development.

There's always time to do it right the second time.

References:

1) 201 Principles of Software Development, Alan Davis

2) www.fastcompany.com/online/06/writestuff.html

3) Barry Boehm, Software Engineering Economics

4) T55 Software Verification Report - Final Report. EDS Corporation.