

Developing a good bedside manner

The fastest and most effective way to debug hardware or software involves using a disciplined process. Here are six steps for debugging.

By [Jack Ganssle](#)
[Embedded.com](#)

(09/23/09, 12:00:00 AM EDT)

Why are debugging code and alien abduction so similar? Because in both cases large blocks of time are unaccounted for.

Medicine, too, resembles debugging (and perhaps alien abduction). In an astonishing development, recently my doctor actually chatted with me for a few minutes. Maybe it was a slow H1N1 day, or perhaps he just wanted a break from all of the runny noses. So I asked him what the hardest part of his job was. Expecting to hear a rant against insurance companies, it was interesting to hear him talk about the difficulty in diagnosing diseases. Most people present with simple cases, but sometimes an individual will have a bewildering array of symptoms that suggest no single etiology. Physicians use differential diagnosis to try and weed causation from the complaints, which is made very complex since patients may ignore some important symptoms while focusing on those that are less critical. To add spice to the sauce of diagnosis, several illnesses may present at the same time.

I was struck by how closely his comments mirror the art of debugging embedded systems. Our nascent product has a bug, which presents itself via some set of symptoms. Press the green button and nothing happens. Is the switch wired incorrectly? Could its Schmidt trigger gate be shot? Is the ISR invoked? Perhaps the ISR never passes a signal to the code that displays a result.

Or maybe the power is off.

In other cases, just as in medicine, one bug may present a variety of odd effects. Or a single symptom could stem from a combination of bugs all interacting in excruciating-complex, and hard to diagnose, manners. I wonder if physicians observe the infrequent symptoms we see, that appear in a system once, go away for weeks, and then randomly resurface?

Then there are the bugs we know exist, but just cannot fix. The system gets shipped with the expectation that sometime someone will see a problem. That, too, is like medicine. "Doc, it hurts when I do this." "Don't do that." My health insurance will not cover kidney stones due to three prior episodes, one of which required an expensive lithotripsy. So that's one latent bug in my gut that will surely reappear at some unknown time in the future.

Stretching the medicine analogy probably too far, the human body swarms with amazing self-repair facilities, rather like a well-designed product sporting watchdog timers, ECC, interlocks, and other protection and recovery mechanisms.

Where medicine and debugging part ways, though, is that in most cases of illness the causes are known. The problem is to pare through the symptoms to correlate them with diseases listed in diagnostics manuals. We have no such manual since there are a huge number of ways a system can fail, and each of those can stem from a staggering number of root causes. So our problem set is vastly greater than the doctor's. Which seems odd, considering that even one of the trillions of cells that make up the human system is astronomically more complex than the biggest embedded system.

But at least we can cut into the patient and change parts till we find the problem. A breakpoint stops its heart; a watchpoint instruments the brain and a recompile chainsaws through any part of the not-quite cadaver. Press "go" and the patient patiently restarts. Dr. Frankenstein could only drool with envy at our god-like powers of reanimation.

Debugging and troubleshooting, which I often use interchangeably since in an embedded system the software and hardware are synergistic, are both art and science. Art, because debugging uses skills built over a lifetime of experience. We draw on similar situations encountered in the past. It's craftsmanship, where much of our skill comes from years of hands-on work. Book learning helps, but like tradespeople, we must serve a long apprenticeship to get a visceral feel for where the clogs typically appear in the plumbing or why the HVAC unit isn't cooling zone three.

Debugging is also science, because these are complex problems that will usually not yield to random efforts. We must be disciplined in how we chase down a bug. Troubleshooting comprises six steps which, in part, neatly parallels the scientific method:

- Observe collateral behavior
- Round up the usual suspects
- Generate a hypothesis
- Invent a fix
- Test the hypothesis
- Apply feedback

First, we observe collateral behavior. The best embedded engineers I've had the pleasure of working with approach their work with a doctor's bedside manner. When a symptom appears, they take their hands off the keyboard and listen very closely to the patient--they look at the symptoms and see what other odd things might be happening. A single bug may manifest itself in several ways, which taken together can help nail it quickly.

They use all of their senses. Does that smell like a burning resistor? Why does the stepper motor now sound faster? That processor never felt quite so hot before!

Next, round up the usual suspects. Too many of us embedded people immediately start setting breakpoints instead of listening to the patient. I see engineers (and have done it myself) wandering into a labyrinth of debugging when the cause is simple. Just this weekend we sailed to Queenstown on Maryland's Eastern Shore and rafted with three other boats. When it was time to go, I fired up Voyager's diesel, but the instruments didn't come on. Out came the schematics, the VOM, and I started tracing wires and taking data. A friend (who happens to be a great embedded systems engineer) said: "Gee, my boat has a circuit breaker on the engine itself. It's hard to get to but could that be the cause?"

It was.

Check the simple stuff, first. And don't rely on assumptions. A very long time ago I was chasing a devilishly convoluted series of problems in my code. The symptoms kept changing. Days went by. My boss, who knew nothing about microcomputers, suggested checking the power supply. I sniggered --what an inane idea! He insisted, and the voltmeter showed a solid 5 volts. It felt good to be vindicated. Until, that is, he put a scope on Vcc and found a high frequency ripple. Swapping power supplies cured all of the myriad problems.

When looking at the system's behavior, remember Bob Pease's adage: "When you find something that looks funny, measure amount of funny." This is engineering. Numbers do tell a story. If the DAC's output is exactly twice as much as anticipated, maybe the input databus has been shifted one bit. Or look for a spurious multiply by two in the code.

Take data. The physician gets your blood pressure and temperature, and perhaps runs an MRI or EKG. Unsubstantiated theories are usually wrong, in science, in medicine, and in debugging. Our experience and analytical abilities suggest what sort of information to mine, using a variety of tools.

Based on the data, generate a hypothesis of what might be wrong.

Dictionary.com defines hypothesis as: "a proposition, or set of propositions, set forth as an explanation for the occurrence of some specified group of phenomena, either asserted merely as a provisional conjecture to guide investigation (working hypothesis) or accepted as highly probable in the light of established facts." In debugging, I prefer to regard a hypothesis as a reasonable, and reasoned, explanation for a system's behavior based on observation of the system's actions and collateral data, such as variable status and info from instruments like scopes and logic analyzers.

A hypothesis is not a guess, it's not a practically random change made in seconds because recompiles are so fast. Alas, too many developers make unreasoned changes in the hope of success. "Heck, let's try this!" Sometimes the patch seems to fix the problem, usually only to create a deeper flaw that silently lurks, awaiting the right (or wrong?) combination of inputs or states to wreak complete havoc.

I advocate the use of engineering notebooks, wherein one writes down a bug's symptom immediately upon encountering it. Take data... and log it in the notebook. Create a hypothesis--and write it down before implementing a fix. The tools have gotten too fast, and have reduced the cost of making a change to nearly zero. That's a wonderful thing compared with the three-day rebuild time we experienced in the early days of the embedded revolution. But it's too often used to displace thinking.

We pay doctors to fix defects. As is often the case for middle-aged folks, for some years my cholesterol levels have been sliding in the wrong direction. Based on his experience, the science, and the data, my doctor came up with a simple fix and prescribed Pravastatin. In the embedded world, too, it's pretty easy to repair a problem once one has a valid hypothesis explaining its root cause. But it's all too easy to use a duct tape solution instead of one that's varnished and highly polished. Industry data shows that for every seven bug fixes, one injects yet another defect, which is about twice the bug rate for writing new code. It doesn't take a rocket scientist or an MD degree to realize one must think deeply about fixing defects.

The lean/agile software development community believes in minimizing waste. Nowhere is this more true than in ensuring bugs are fixed correctly. Alas, I see too many agilists (and plenty of others) fill a virtual landfill with waste by being hasty and thoughtless when dealing with defects. Refactoring is a great concept that is all too often neglected in the madness of a release.

Sometimes, sadly, one must do horrible things to the code to get a system out the door. Luke Hohmann advises "post-release entropy reduction." Entropy is the measure of disorder in a closed system, and in software things get worse--more entropic--from version to version as crappy changes accumulate. Hohmann suggests that if one has to make a hurried and perhaps somewhat shoddy change, the very first task when starting on the next is to refactor that abomination.

I'm due for a blood test to see if the Pravastatin has brought down my LDL levels. In other words, my physician wishes to test his hypothesis about the choice of drugs. Too many of my friends are on chemotherapy for cancer--they all get MRIed after the course of treatment to see if the chemo worked. The same goes for debugging. Implement the fix and test your hypothesis. Run the tests again, thoroughly, to be sure the problem is really gone, regardless of inputs or other stress that might be applied to the system.

Finally, apply feedback. What did you learn from this problem? Could other parts of the system suffer from the same flaw? This summer swine flu nearly put a pal prematurely in the grave. The medical community looked into his travel and contacts with others to ensure that none of them had been infected. Fixing broken code is great; better is to use the bug as a chance to find other flaws. And use it to improve your validation suite. In regression testing, we ensure that for every problem uncovered there's a test, in case the same bug resurfaces.

The fastest and most effective way to debug hardware or software involves using a disciplined process. I've described mine; others exist. One of the best is from Steve Litt, who manages www.troubleshooters.com. His book *Twenty Eight Tales of Troubleshooting* (available from his web site) is both evocative and highly entertaining. I literally couldn't put it down.

Jack Ganssle (jack@ganssle.com) is a lecturer and consultant specializing in embedded systems' development issues. For more information about Jack [click here](#).