HOW TO DEBBUG

One of the painful parts of teaching a lab-based embedded systems course is that over and over I have to watch a team with a relatively simple bug in their code, but who is trying to fix it by repeatedly making random changes. Generally they start with code that's pretty close to working and break it worse and worse. By the end of the lab they're frustrated, aren't any closer to finding the bug, and have made a complete mess of their code, forcing them to go back to the previous day or week's version.

A typical Computer Science curriculum fails to teach debugging in any serious way. I'm not talking about teaching students to use debugging tools. Rather, we fail to teach the thing that's actually important: how to think about debugging. Part of the problem is that most CS programming assignments are small, self-contained, and not really very difficult. The other part of the problem is that debugging is not addressed explicitly. After noticing these problems I started to focus on teaching students how to debug during lab sessions and also made a lecture on debugging that I give each year; this piece elaborates on that lecture.

First we'll want to define some terms:

- Symptom — a faulty program behavior that you can see, such as crashing or producing the wrong answer
- Bug — a flaw in a computer system that may have zero or more symptoms
- Latent bug — an asymptomatic bug; it will show itself at an inconvenient time
- Debugging — using symptoms and other information to find a bug
- Failure-inducing input — input to the program that causes the bug to execute and symptoms to appear. In some cases, capturing the complete failure-inducing input is hard because it may include window system events, hardware-level events like

interrupts or bit-flips in RAM cells, and OS-level events like context switches and TCP timeouts. The times at which elements of the input occur may be important.

- Deterministic platform — A platform having the property that it can reliably reproduce a bug from its failure-inducing input. Simulators should be deterministic, but getting determinism from bugs involving hardware/software interactions may be difficult or impossible.

The high-level reason debugging is hard is that it's an inverse problem: it attempts to infer the cause for observed effects. Inverse problems are generally ill-posed and extra input is required to find a unique solution. A debugging problem may be especially difficult if:

- A deterministic platform is unavailable
- The bug is costly to reproduce, for example because it takes a long time to show up, it requires expensive hardware, or it requires many machines
- Visibility into the executing system is limited, for example because it involves an embedded processor that doesn't have printf() or JTAG
- Several distinct bugs are collaborating to produce the symptoms that you see
- The bug and its symptoms are widely separated in space and time
- The bug is actually a mistaken assumption made by developers — How many times have you been burned because you were running an old version of the code, had some environment variable wrong, were linking against the wrong library, etc.?
- The bug is in a part of the system considered out of scope: hardware, operating system, compiler, etc.

Of course, a very bad bug will involve several of these factors at the same time. A nice hard bug can set a development team back by weeks or more. The key to avoiding this kind of delay is to approach debugging with the correct mindset and with a good deal of experience. Careful, focused thinking can often see through a hard bug, whereas unfocused thinking combined with random changes to the program will not only waste time, but probably also break parts of the system that previously worked.

A Scientific Approach to Debugging

The following steps constitute a fairly complete approach to debugging. If I've done a good job describing these steps, as you read them you'll see that you do, in fact, perform most of them every time you debug, but in a very informal and implicit way. The goal here is to make the entire process explicit not just to clarify our understanding, but also because I believe that the worst-case bugs — those that threaten to stop you for weeks or more — are best met with a methodical, deliberate approach.

1. Verify the Bug and Determine Correct Behavior

It makes no sense to even start debugging unless we're pretty sure:

1. There is actually a bug.
2. We understand what the system should have done.

Sometimes these questions are easy to answer ("robot wasn't supposed to catch fire") but other times they can be distressingly difficult. I've gotten into long, drawn out arguments with smart people about whether a particular behavior exhibited by a C compiler is a bug or not; you'd think this wouldn't happen for a language with a 550-page standard. For a computer system where no specification or reference implementation exists — for example, a new supernova simulation — we may have no idea whatsoever what the correct answer is.

2. Stabilize, Isolate, and Minimize

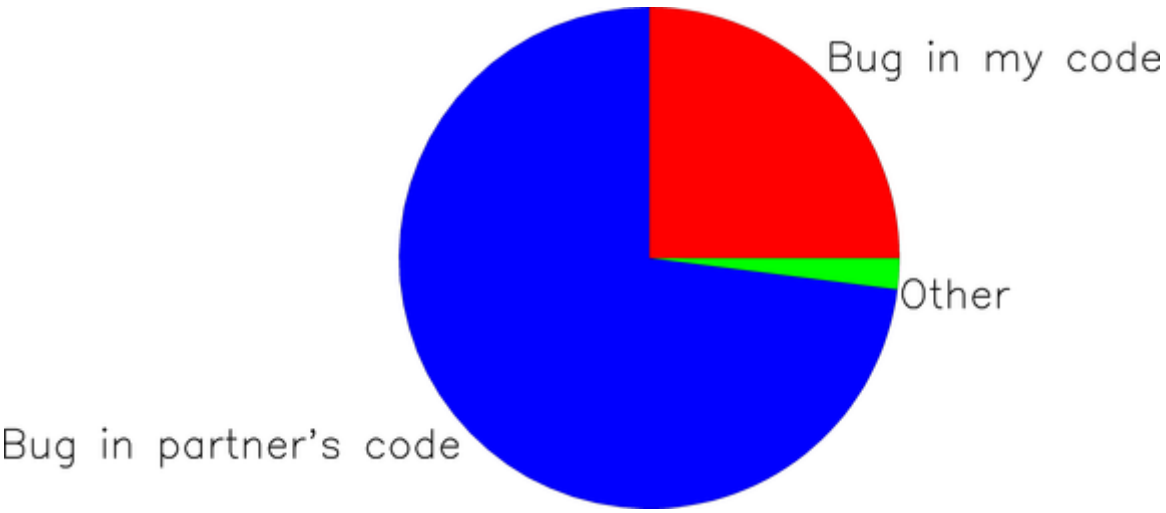
Now that we're sure we have a real bug, we want to be able to reproduce it deterministically by isolating its failure-inducing input. If this input is timing-dependent or includes hardware-level events, the situation may become challenging and we need to stop and think: Is there some clever way to make the system deterministic? For example, let's say that we have some bizarre interaction between an interrupt handler and our main loop. If we know about where the interrupt is firing, it may be possible to just disable the actual interrupt and insert a call to the

interrupt-handling function from our main loop. If this works, our life is a lot easier. If not, we may still learn something.

This is a good time to make the failure-inducing input smaller if at all possible. For example, if our robot software crashes two hours into some sequence of activities, can we make it crash faster by moving the failing activity earlier in the robot's schedule? If our word processing application crashes when it loads some huge document, can we find a single-page document that still crashes it by splitting the big doc into individual pages? Small failure-inducing inputs are important for two reasons. First, they are easier to work with and save time. Second, it is usually the case that larger failure-inducing inputs contain more junk that is irrelevant to the problem at hand, making it harder to spot the pattern that triggers the bug. If you can find a truly small failure-inducing input, often you'll have done most of the debugging work without looking at a line of code.

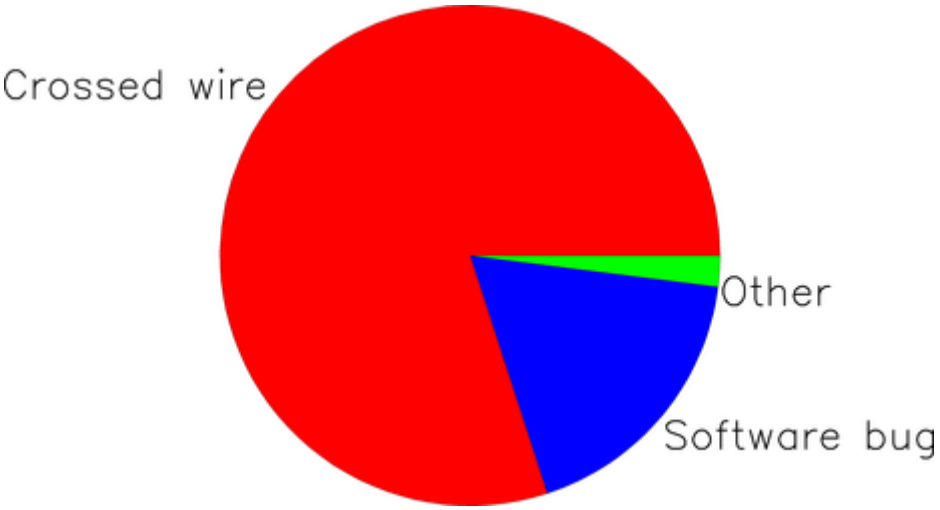
3. Estimate a Probability Distribution for the Bug

Now we move from science to art. Using all available information, we have to guess what bug is causing our symptoms. Our first guesses can be crude; we'll have plenty of time to refine them later. Let's take a simple example where my partner and I are working on an embedded systems project and it doesn't work: the robot moves left when it should move right. Since my partner implemented the finite state machine that controls robot direction, I'm going to guess this is her bug, while conceding the possibility that my own code may be at fault:



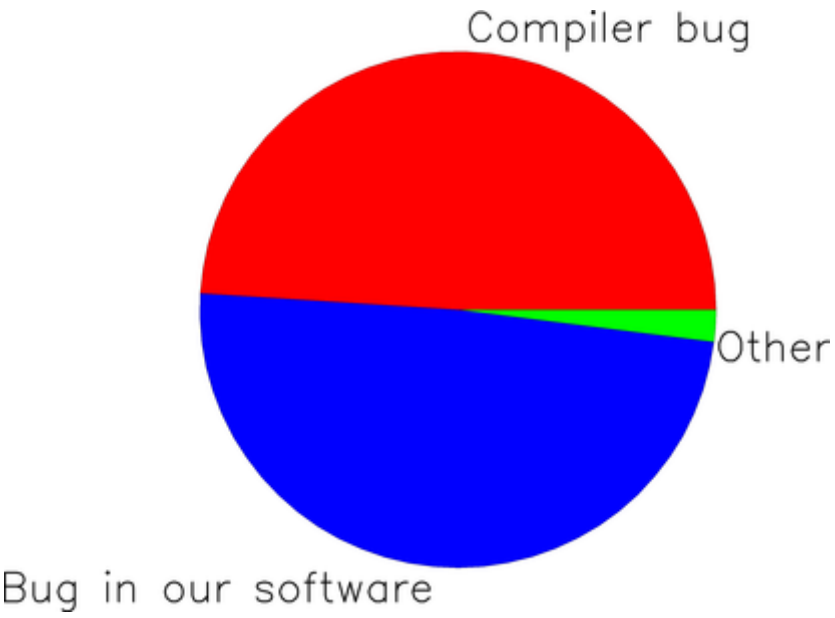
The interpretation of this pie chart is "I estimate about a 75% chance that the bug is in my partner's code, a 25% chance it's in my code, and a small chance it's somewhere else entirely."

Next, we'll ask our instructor to take a guess. It turns out he's been walking around the lab all afternoon looking over teams' shoulders and has already seen three groups whose robot went the wrong direction because they wired up their motor driver chip the wrong way due to an ambiguity in the hardware documentation. Based on this experience, he estimates the bug's probability distribution as:



Finally, let's say that my partner knows something the rest of us don't: she just finished an embedded project which used the same compiler that we're using in class, which happens to have an extremely buggy implementation of the square root

function. Furthermore, we have been using a lot of square root operations in our distance-estimation module and elsewhere. Thus, her estimation of the bug's probability distribution is:



Does any of us know where the bug actually is? Of course not — this is all pure guesswork. But good guesses are important because they feed the next step of the process. Where do these probability distributions come from in practice? From years of bitter experience with similar debugging situations! It's important to approach this step with an open mind; if the true location of the bug is not represented anywhere in our probability distribution, we're going to waste a lot of time.

When guessing where the bug lives, it's worth keeping Occam's Razor in mind:

When you have two explanations, both consistent with the available evidence, prefer the simpler one.

Before moving on let's notice that by talking to several people, we've failed to meet the original goal of estimating the probability distribution for this bug: we have three distributions, not one. To fix this we could combine these three distributions, perhaps giving higher weight to whoever we trust the most. In practice, however, this isn't going to be necessary since the separate distributions have given us enough information that we can proceed.

Why did we talk to several people in the first place? It's because there's an insidious problem that underlies a lot of debugging woes. If I have an error in my thinking that leads to bugs, this same error also makes it very difficult for me to figure out where the bug is. Programming in teams is a good start, but isn't a complete answer because teams can easily talk themselves into all thinking the wrong way. A fresh set of eyes — a person who was not closely involved with the system's design and implementation — is always useful.

4. Devise and Run an Experiment

Our next step is to eliminate some of the possibilities by running an experiment. Ideally, we'll think of an experiment with a yes/no result that divides the probability space in half. That is, half of the possible bugs are eliminated regardless of the experiment's result. In an information-theoretic sense, we maximize the information content of the experiment's result when it splits the space of possibilities into parts of equal size. A few examples will help make this concrete.

Let's start with my probability distribution from the first pie chart above. The obvious hypothesis is that my partner's FSM is faulty. To test this hypothesis, we instrument her FSM with lots of assertions and debugging printouts (assuming these are possible on our robot) and run it for a while. If we did things right, the results of this experiment will either support or refute the hypothesis.

The second probability distribution, the one our instructor came up with, leads to an entirely different experiment. Here the obvious hypothesis is that our robot control board is wired improperly. To test this, we'll either closely inspect our wiring or else move our robot software over to a different robot that is known to be wired up correctly. Either way, the hypothesis will be rapidly validated or falsified.

The third probability distribution leads to the hypothesis that the compiler is buggy. To test this, we'll compile our robot software using a different compiler (we should be so lucky that this is easy…) and see if the program behavior changes.

Which of the three experiments should we run first? It probably doesn't matter much — we should choose either the easiest experiment or else the one whose proponent has the strongest feelings.

Hopefully the pattern is now clear: based on our intuition about where the bug lies, we design an experiment that will reject roughly 50% of the possible bugs. Of course I'm oversimplifying a bit. First, experiments don't always have yes/no results. Second, it may be preferable to run an easy experiment that rejects 15% of the possible bugs rather than a difficult one that rejects exactly 50% of them. There's no wrong way to do this as long as we rapidly home in on the bug.

5. Iterate Until the Bug is Found

Steps 3 and 4 need to be repeated until finally we devise an experiment whose result serves as the smoking gun that unambiguously identifies the bug. This iteration leads us to perform an approximate binary search over the space in which we believe the bug to live. If our estimation of the bug's probability distribution is accurate, this lets us find the bug by running a number of experiments logarithmic in the number of possible bugs.

6. Fix the Bug and Verify the Fix

Fixing is usually the easy part. Then, run the entire test suite to help ensure that the bug was really fixed and that the fix didn't break anything that used to work.

7. Undo Changes

Finding and fixing a showstopping bug is a huge relief: finally you and the rest of your team can start to make progress again. However, in the rush to get back to work it's easy to forget to undo any changes made to the code base to support debugging such as turning off the compiler's optimizer, inserting extra printfs, and stubbing out time-consuming routines. It is important to roll back all of these changes or you'll get nasty surprises later.

8. Create a Regression Test

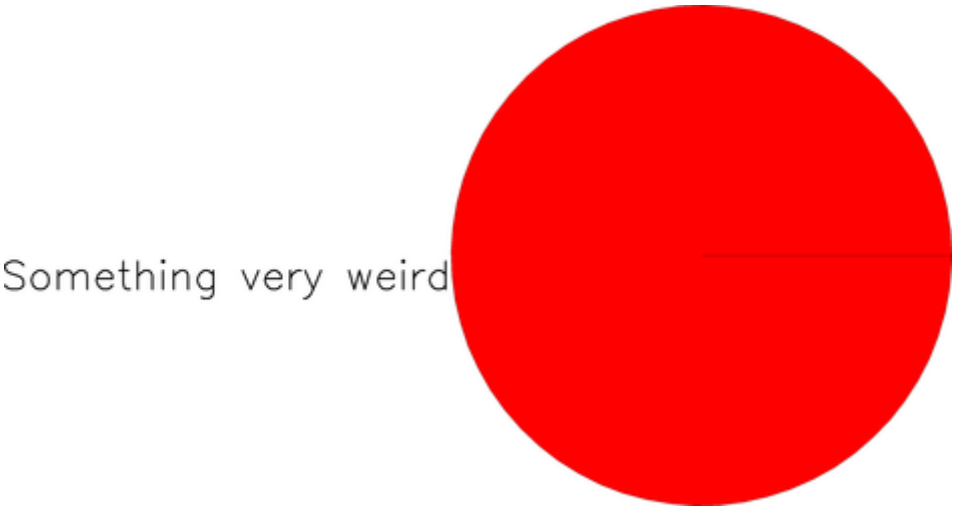
Add a test case to the test suite which triggers the bug that was just fixed.

9. Find the Bug's Friends and Relatives

Earlier I mentioned that debugging is an inverse problem. There's a second inverse problem — which may be easier or harder than the first — which is to figure out what error in thinking led to the bug. This is important because the same thinko probably caused several bugs and you've only found one of them. For example, if we find a bug where I failed to properly check the return value from some library function, it's a good bet that I did the same thing elsewhere. If we quickly inspect each library call in the code I've written, we'll have a good chance of finding these. Of course we don't have to do this, but if we don't, the latent bugs will sit there waiting to break the system.

What If You Get Stuck?

In the examples above, each person's probability distribution for the bug naturally suggested an experiment to run. This is not always how it works. If you are inexperienced or if the bug involves something relatively nasty like a difficult race condition or memory corruption, at some point in the debugging session your probability distribution may end up looking like this:



In other words, we're 100% sure that we have no idea what is going on. Obviously you want to avoid this situation because life is now difficult: scientific debugging has reached an impasse. Your options aren't great but there are still plenty of things to try:

- Spend more time attempting to reduce the size of the failure-inducing input; often this gives insights into the nature of the problem
- Gain additional visibility into the system by adding more watchpoints or debug print statements
- Find a tool that can bring new information to light — valgrind, better compiler warnings, a race condition checker, or similar
- Take a step back and question your assumptions about this bug; talking to someone new or just taking a walk can both help
- Look at the code and think until you see the answer

Somehow, at some point in every serious programming project, it always comes down to the last option: stare at the code until you figure it out. I wish I had a better answer, but I don't. Anyway, it builds character.

Another thing to keep in mind when you're stuck is that perhaps you're looking for interesting bugs, but the actual problem is boring and stupid. For example, the makefile is missing the dependency that's supposed to rebuild the code we're modifying, or a file is not being created because the disk quota is exceeded. Since there are a lot of things like this that can go wrong, it's not possible to foresee them all. A reasonable way to deal with them is to develop a quick set of heuristics to try when system behavior feels funny, such as:

- Run "make clean"
- Logout and log back in
- Reboot the machine
- Run commands from someone else's account
- Use a different machine

Although it often feels wrong to resort to silly activities like these, they can really save time by rapidly ruling out broad categories of improbable problems.

Stupid quota problems and the like are difficult to debug for a simple reason: You, the developer, are heavily invested in making the program logic correct. This is hard work and it occupies nearly all of your brain power. When a bug pops up, the natural assumption to make is that it's due to a flaw in your logic. This is often, but not always, the case.