The Ganssle Group

Perfecting the Art of Building Embedded Systems

The Ganssle Group
PO Box 38346, Baltimore, MD 21231
Email info@ganssle.com

Troubleshooting

Troubleshooting skills are critical to debugging embedded systems. Sure, a lot of these skills come only from experience. Many, though, are more of a philosophical nature, as described in the following.

Published in EDN in October, 1994.

There comes a time in any project when your new design is finally assembled, awaiting your special expertise to "make it work". Sometimes it seems like the design end of this business is the easy part; troubleshooting prototype hardware can make even the toughest engineer a Maalox and Rogaine addict.

You can't fix any embedded system without the right world view; a zeitgeist of suspicion tempered by trust in the laws of physics, curiosity dulled only by the determination to stay focused on a single problem, and a zealot's regard for the scientific method.

Perhaps these are successful characteristics of all who pursue the truth. In a world where we are surrounded by complexity, where we deal daily with equipment and systems only half-understood, it seems wise to follow understanding by an iterative loop of focus, hypothesis, and experiment.

Too many engineers fall in love with their creations only to be continually blindsided by the design's faults. They are quick to overtly or subconsciously assume the problem being chased is due to the software, the lousy chips, or the power company, when simple experience teaches us that any new design is rife with bugs.

Assume it's broken. Never figure anything is working right until proven by repeated experiment; even then, continue to view the "fact" that it seems to work with suspicion. Bugs are not bad; they're merely a test of your troubleshooting ability.

Armed with a healthy skeptical attitude, the basic philosophy of debugging any system is to follow these steps:

```
For (i=0; i< # findable bugs; i++)
  {
  while (bug(i))
    {
    Observe the behavior to find the apparent bug;
    Observe collateral behavior to gain as much information as possible
about the bug;
```

```
    Round up the usual suspects;
    Generate a hypothesis;
    Generate an experiment to test the hypothesis;
    Fix the bug;
    };
  };
```

Now you're ready to start troubleshooting, right? Wrong! Stop a minute and make sure you have good access to the system. No matter how minor the problem seems to be, troubleshooting is like a bog we all get trapped in for far too long. Take a minute to ease your access to the system.

Do you have extender cards if they're needed to scope any point on the board(s)? How about special long cables to reach the boards once they are extended?

If there's no convenient point to *reliably* clip on the scope's ground lead, solder a resistor lead onto the board so you're not fumbling with leads that keep popping off.

Some systems have signals that regulate major operating modes. Solder a resistor lead on these points as well, as you'll surely be scoping them at some point. This small investment in time up front will pay off in spades later.

Following the Loop

Let's cover each step of the troubleshooting sequence in detail.

Step 1: *Observe the behavior to find the apparent bug*.

In other words, determine the bug's symptoms. Remember always that many problems are subtle and exhibit themselves via a confusing set of symptoms. The fact that the first digit of the LCD fails to display may *not* be a useful symptom -- but the fact that none of the digits work may mean a lot.

Step 2: *Observe collateral behavior to gain as much information as possible about the bug* .

Does the LCD's problem correlate to a relay clicking in? Try to avoid studying a bug in isolation, but at the same time be wary of trying to fix too many bugs at the same time. When ROM accesses are unreliable and the front panel display is not bright enough, address one of these problems at a time. No one is smart enough to deal with multiple bugs all at once - unless they are all manifestations of something more fundamental.

Step 3: *Round up the usual suspects.*

Lots of computer problems stem from the same few sources. Clocks must be stable and must meet very specific timing and electrical specs... or all bets are off. Reset too often has unusual timing parameters. When things are just "weird", take a minute to scope all critical inputs to the microprocessor, like clock, HOLD, READY, RESET, and the like.

Never, never, never forget to check Vcc. Time and time again here at Softaid we see systems that don't run right because the 5 volt supply is really only putting out 4.5... or 5.6... or 5 volts with lots of ripple. The systems come in after their designers spent weeks sweating over some obscure problem that in fact never existed, but was simply the ghostly incarnation of the more profound power supply issue.

Step 4: *Generate a hypothesis.*

"Shotgunners" are those poor fools who address problems by simply changing things - ICs, designs, PAL equations - without having a rational for the changes. Shotgunning is for amateurs. It has no place in a professional engineering lab.

Before changing things, formulate a hypothesis about the cause of the bug. You probably don't have the information to do this without gathering more data. Use a scope, emulator, or logic analyzer to see exactly what *is* going on; compare that to what you think *should* happen. Generate a theory about the cause of the bug from the difference in these.

Sometimes you'll have no clue what the problem might be. Scoping the logical places might not generate much information. Or, a grand failure like an inability to boot is so systemic that it's hard to tell where to start looking. Sometimes, when the pangs of desperation set in, it's worthwhile to scope around the board practically at random. You might find a floating line, an unconnected ground pin, or something unexpected. Scope around, but be always on the prowl for a working hypothesis.

Step 5: *Generate an experiment to test the hypothesis.*

Construct an experiment to prove or disprove your hypothesis. Most of the time this gets resolved in the process of gathering data to come up with the theory in the first place. For example, if the emulator reads all ones from a programmed ROM, a reasonable hypothesis is that CS or OE is not toggling. Scoping the pins will prove this one way or the other, though now you'll need another hypothesis and experiment to figure out why the selects are not where you expect to see them.

Sometimes, though, the hypothesis-experiment model should be much less casually applied. When Intel started shipping the XL version of the 186 (supposedly compatible with the older series), we found that none of our systems worked. Scoping around showed the processor to be stuck in a weird tri-state, though all of it's inputs seemed reasonable. One hypothesis was that the 186XL was not coming out of reset properly, an awfully hard thing to capture since reset is a basically non-scopable one-time event. We finally built a system to reset the processor repeatedly, to give us something to scope. The experiment proved the hypothesis, and a fix was easy to design.

Note that an alternative would have been to glue in a new reset circuit from the start to see if the problem went away. Problems that mysteriously go away tend to mysteriously come back; unless you can prove that the change really fixed the problem, there may still be a lurking time bomb in the system.

Occasionally the bug will be too complicated to yield to such casual troubleshooting. If the timing of a PAL will have to be adjusted, before wildly making changes visualize the new timing in your mind or on a sheet of graph paper. Will it work? It's much faster to think out the change than to actually implement it... and perhaps troubleshoot it all over again.

Rapid troubleshooting is as important as accurate troubleshooting. Decide what your experiment will be, and then stop and think it though once again. What will this test really prove? I like experiments with binary results - the signal is there or it is not, or it meets specified timing or it does not - since either result gives me a direction to proceed. Binary results have another benefit: sometimes they let you skip the experiment altogether! Always think through the actions you'll take *after* the experiment is complete, since sometimes you'll find yourself taking the same path regardless of the result, making the experiment superfluous.

If the experiment is a nuisance to set up, is there a simpler approach? Hooking up 50 logic analyzer probes is rather painful if you can get the same information in some easier way. I'd hate to be in a lab with a logic analyzer since they are so useful for so many things... but I try to relegate it to the tool of last resort, since most often it's possible to construct an easier experiment that is complete in a fraction of the time it takes to connect the LA.

Don't be so enamored of your new grand hypothesis that you miss data that might disprove it! The purpose of a hypothesis is simply to crystallize your thinking - if it is right, you'll know what step to take next. If it's wrong, collect more data to formulate yet another theory.

Step 6: *Fix the bug.*

There's more than one way to fix a problem. Hanging a capacitor on a PAL output to skew it a few nanoseconds is one way; another is to adjust the design to avoid the race condition entirely.

Sometimes a quick and dirty fix might be worthwhile to avoid getting hung up on one little point if you are after bigger game. Always, always, revisit the kludge and re-engineer it properly. Electronics has an unfortunate tendency to work in the engineering lab and not go wrong until the 5,000th unit is built. If a fix feels bad, or if you have to furtively look over your shoulder and

glue it in when no one is looking, then it is bad.

Finally, never, ever, fix the bug and assume it's OK because the symptom has disappeared. Apply a little common sense and scope the signals to make sure you haven't serendipitously fixed the problem by creating a lurking new one.

Other Ideas

Constantly apply sanity checks. Twenty years ago the Firesign Theater put out an album called "Everything You Know is Wrong". Use that as your guiding philosophy in troubleshooting an embedded system. For example, just because you checked Vcc last night and it was fine does not mean that it's OK now. Prototype systems fail in wondrous ways, so always be on the lookout.

Another example: suppose your system runs fine at 10 Mhz but never at 20. Obviously you'd put a 20 Mhz clock source in a pursue the problem. Every once in a while go back to 10 Mhz just to be sure the symptom has not changed. You could spend a lot of time developing hypothesis about 20 Vs 10 operation, when the 10 Mhz test results might actually be a fluke.

It's a good idea to be on the lookout for excessive heat, especially now that so many components are surface mounted and tough to change when you blow them up.

All semiconductor devices generate some heat; big CPUs can produce quite a bit. A really hot device, one that you can't keep your finger on, is usually screaming for help. Excessive heat may indicate an SCR latch up condition due to outrageous ground bounce or a floating input.

Less dramatic overheating, much harder to detect without a lot of practice, often indicates a design flaw. Your finger can give important clues about the design. If two devices try to drive the bus at the same time, they'll overheat.

Be careful how you apply your personal temperature sensor. I've found that my callused forefinger is insulated enough to protect me from bad burns when a part is unexpectedly frying. Thus, I gingerly tough each part; if it seems reasonably cool I'll then use the much-more-sensitive back of my hand to try and determine if the chip is running hotter than it should. It's surprising how much information you can get with a little experience.

Final Words

At 3:00 AM when the problems seem intractable and you're ready to give up engineering, remember that the system is only a computer. Never panic - you are smarter than it is.