Last Time

- Advanced interrupt issues
  - ColdFire interrupts
  - System design
  - Prioritized interrupts
  - Interrupt latency
  - Race conditions
  - Reentrancy
  - Interrupt overload
  - Missed interrupts
  - Spurious interrupts

Today

- Debugging embedded software
  - The best debuggers are probably 100x more effective than average ones
  - The worst debuggers are probably 100x worse than average ones
  - You can learn to be one of the better ones

- Your friend calls you and says "Oh no – the lamp in my room is all dark!"
- How do you help?

Debugging

- Today is about strategies for humans
  - Talked about tools a few weeks ago
- Lecture contents
  - Definitions
  - Good debugging techniques
  - Ways to avoid debugging
- Why do we care?
  - The most difficult bugs that you'll run into will cannot be found casually
    - Systematic and thorough approach is required
  - Probably 100X difference in effectiveness between good and bad debuggers

---

Debugging is twice as hard as writing the code in the first place. Therefore, if you write code as cleverly as possible, you are, by definition, not smart enough to debug it.

– Brian Kernighan

An Early Computer Bug

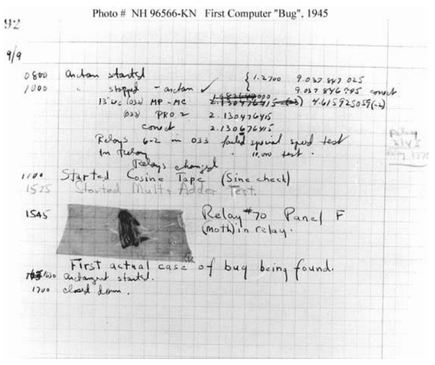


Photo # NH 96566-KN First Computer "Bug", 1945

92

9/9

0800 Arctan started
1000 " stopped – arctan ✓ {1.2700 9.037 847 025 / 9.037 846 995 const
13" UC (032) MP – MC ~~1.982147000~~ ~~2.130476415~~ (-3) 4.615925059(-2)
(033) PRO 2 2.130476415
const 2.130676415
Relays 6-2 in 033 failed special speed test
In relay . 11,000 test .
Relays changed
1100 Started Cosine Tape (Sine check)
1525 Started Mult + Adder Test.
1545 Relay #70 Panel F
(moth) in relay.
First actual case of bug being found.
1630 arctangent started.
1700 closed down.

Definitions

- Symptom – Something wrong that you can see
- Bug – Software error that causes a symptom
- Debugging –
  1. Finding the bug that causes a symptom – hard!
  2. Fixing the bug – usually not too hard
- Failure-inducing input – Input that causes a bug to execute
  - Complication: The time at which events occur must be considered part of the input
    - E.g. it can matter when an interrupt fires
    - Hard to reproduce this

More Definitions

- Deterministic system – One where failure-inducing inputs can be reproduced
  - Nondeterministic systems are much harder to debug
  - Sometimes non-deterministic systems can be turned into deterministic ones
  - Simulators are generally deterministic

Debugging is harder when…

- Visibility into the executing system is limited
- Several bugs are collaborating to cause the symptom that you see
- The bug and symptom are widely separated in space or time
- The "bug" is a mistaken assumption
- The bug is in a library, hardware, OS, or compiler
- The system is nondeterministic
- Worst case scenario: Many of these complications are present at the same time
  - You will need to stare at the code very hard
  - Or throw it all away and start over

Ugly Fact

- Often the bug is something so stupid that you'll have a hard time thinking of it
  - Accidentally running an old version of the software
  - Environment variable (e.g. PATH) is hosed
  - Buggy Makefile resulted in a file not being recompiled
  - Software upgrade added a DLL that is buggy
  - File not created since disk is full or you're over quota
  - File unavailable due to network glitch
- There is an infinite variety of these – develop quick sanity checks to rule them out
  - Rebuild entire application
  - Reboot or switch to a different machine
  - Etc.

Rest of this Lecture

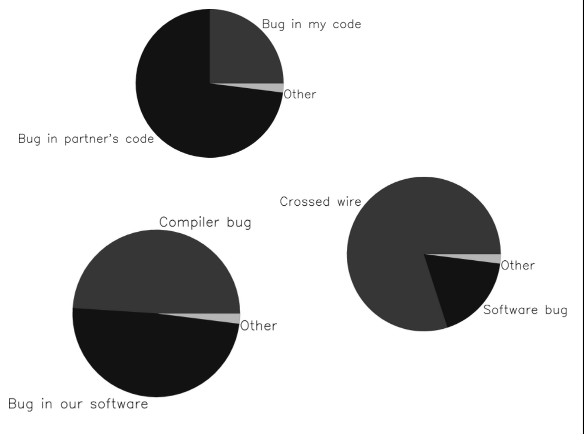
1. Scientific debugging
   - Debugging is just a binary search
2. Techniques for avoiding debugging

Scientific Debugging

- Step 1: Verify the bug
  - Make sure it's really a bug
  - Make sure we understand what was supposed to happen
  - Often this step is easy, but sometimes very tricky
    - Sometimes we don't know correct behavior for the situation
- Step 2: Stabilize, isolate, and minimize
  - Buggy behavior must be repeatable (not necessarily deterministic)
    - Pin down system inputs and as many other variables as possible
  - Try to reduce the size of the failure-inducing input

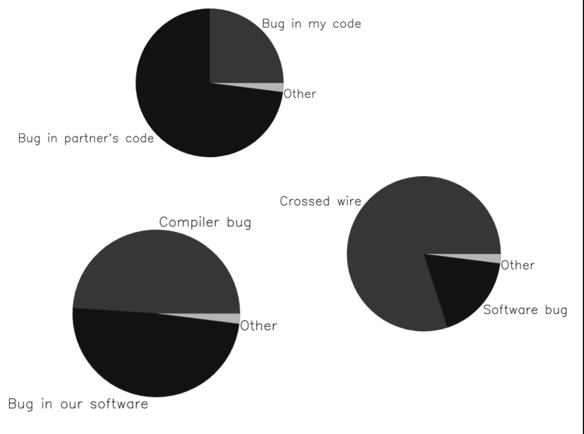
Scientific Debugging

- Step 3: Estimate where the bug is
  - Make educated guesses about causes that might explain the observed behavior
  - Initial hypothesis is often pretty vague – this is ok
  - Write down the hypothesis
  - Rank possibilities in order of likelihood
  - This step provides structure for your bug hunt



Scientific Debugging

- Step 4: Devise and run an experiment
  - Should be designed to reject ~50% of the bug's probability space
  - However, prefer simple experiments over difficult / complicated ones



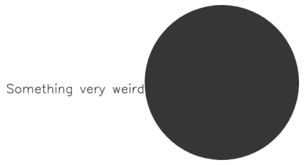
Scientific Debugging

- Step 5: Go back to Step 4 until you find the bug
- Debugging by binary search
  - Requires log N experiments if there are originally N hypotheses

Scientific Debugging

- Step 6: Fix the bug and verify the fix
- Step 7: Undo any changes introduced during debugging
  - Easy to forget this!
- Step 8: Create a regression test
- Step 9: Find the bugs friends and relatives

What if you're stuck?



- Spend more time reducing the input
- Get better visibility into the executing system
  - Find a better HW or SW tool

More getting unstuck…

- Rethink your assumptions
  - If a bug you wrote is due to a conceptual error, then by definition you can't find the bug
  - This is common in practice
- Talking it over can help
- This happens a lot:
  - Mail #1 – Dr. Regehr I can't get my system to work…
  - Mail #2 (15 minutes later) – Never mind I found it…
- Stare hard at the code and think
  - It always comes down to this at some point

More getting unstuck

- Read the manual again
- Try a different board
- Run "make clean"
- Use someone else's account
- Log out and log back in
- Reboot the machine

Incremental Development

- Starting with working code:
  1. Make small change
  2. Test
  3. If it still works, go to step 1
  4. If it doesn't work, back out the change and to go step 1
- Why don't we always do this?
  - We think we can get it right
  - Incremental development is a hassle
- Failure to develop incrementally is one of the most common errors I see while watching students

Defensive Programming

- Reduce code complexity
  - Don't optimize until you know it's needed
- Test early and often
- Avoid hidden dependencies
  - E.g. writing an accelerometer driver that assumes the timer is already initialized
- Be very careful when reusing code
  - Write test cases
  - Check values that come from external code
- Include sanity checks
  - Assume your code runs in a hostile environment
- Don't ignore failures and errors

Programming with assert()

- Sanity check: Making sure something that "must be true" is actually true
- Placing asserts well requires judgement calls
- Bad assert:

```
y = x + z;  assert (y == (x + z));
```

- Good asserts:

```
assert (x >= 0);  y = sqrt(x);
assert (p);  *p = 5;
assert (i < 10);  a[i]++;
assert (sizeof(int) == 4);
x = malloc(sizeof(foo));  assert ((x & 0x7) == 0);
function_that_doesnt_return();  assert (0);
assert (SP > (_bss_end + 75));
```

More assert()

- Good asserts check
  - Preconditions – must be true for code to run successfully
  - Postconditions – must be true after code runs
  - Invariants – must be true at all times (except while data structure is being manipulated)
    - E.g. length field of a linked list ADT stores the length
- Some assertions are "static" – they can be resolved when your system is compiled
  - Compiler support for these would be nice
- Assertions are not for runtime error handling
  - Failed assert == bug
- Usually leave assertions turned on in production code

Delta Debugging

- Idea: Searching for some kinds of bugs can be automated!
- Requires
  - A way to run the program and automatically determine if a bug was encountered
  - A deterministic system
- Delta algorithm:
  1. Run the program on input that makes the bug happen
  2. Delete some of the input
  3. See if the bug still happens
     - If yes, keep the new input and go to step 1
     - If no, go back to old input and go to step 1
- Result: Smaller input that makes the bug happen

Conclusions

- Debugging is harder than programming
- Major debugging strategies
  - Scientific debugging
  - Incremental development
  - Defensive programming
- When confronted with a hard bug
  - First impulse is to start hacking
  - Rather, stop and think
- Being good at debugging will save you huge amounts of time