Course "Debugging"

Debugging – An Introduction

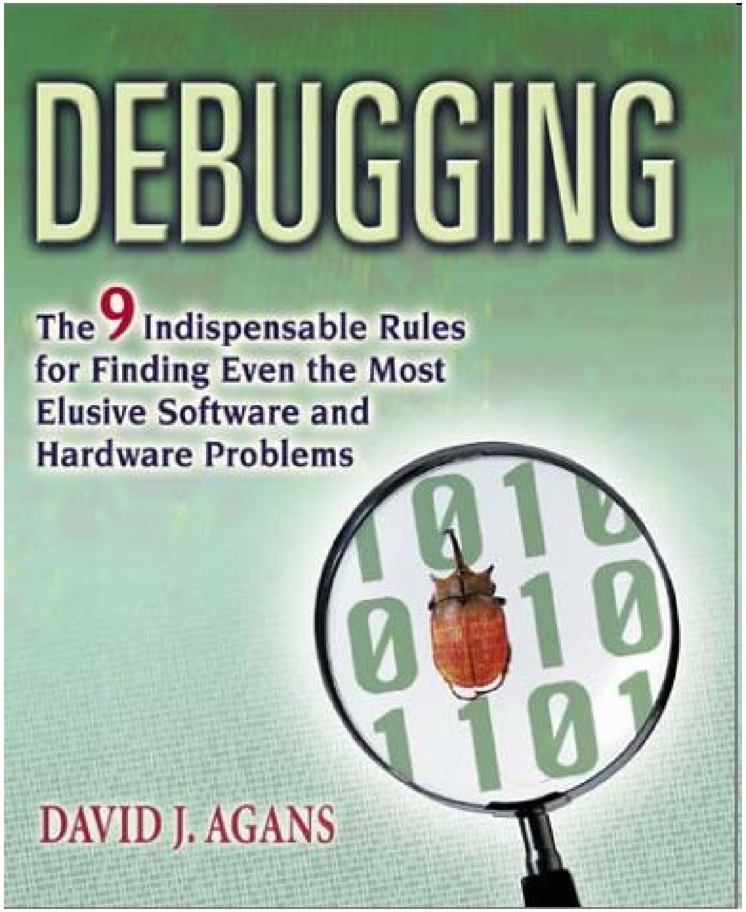
Prof. Dr. Lutz Prechelt
Freie Universität Berlin, Institut für Informatik
http://www.inf.fu-berlin.de/inst/ag-se/

- Universal method of debugging
- 9 rules with subrules and examples ("war stories")

- Rule 1:
  - Understand the system
- Rule 2:
  - Make it fail
- Rule 3:
  - Quit thinking and look

Source

- David J. Agans:
  "Debugging – The 9 indispensable rules for finding even the most elusive software and hardware problems", Amacom, NY, 2002
  - Only 180 pages
  - A fun read!



Proven Rules

- We will cover 9 generic methodological rules to be used during debugging
- Derived from long experience
- Proven to work:
  - Engineers quickly improve in their debugging skills when they learn to use these rules
  - Most engineers already apply most of the rules
    - more or less precisely, consciously, and consequently

Obvious?

- Most of these rules may look obvious

BUT:

- Obvious does not mean easy
- It is not obvious how to apply them to any one problem
- Often neglected in the "heat of the battle"
- Few people follow all of these rules naturally
  - "Debugging is an art"

Universal

The rules work for

- software
- computer hardware
- other electronics
- cars
- houses
- human bodies
- etc.

- We will focus on software
  - (and more on systems rather than code)
  - but use many examples from other areas as well

The rules work if the system

- has been designed wrong
- has been built wrong
- has been used wrong
- is broken

What the rules are about

The purpose of the rules is to

- help determine the causes of misbehavior (defects)
- help correct the causes of misbehavior

The purpose is <u>not</u> to

- prevent defects ("process management")
- detect the presence of defects ("testing", "use")
- decide whether a defect should be corrected (an aspect of "quality management")

Debugging terminology

(For software only:)

- First the programmer does something wrong, or fails to do something that is required
  - This is called an **error**
  - Errors are events and are performed by humans
- As a result of the error, the software may have incorrect structure
  - This is called a **defect** (or fault)
- As a result of the defect, the software may behave incorrectly when executed
  - This is called a **failure**
- During testing or use, we observe failures and conclude there must be a defect
- **Debugging is primarily about locating the defect**

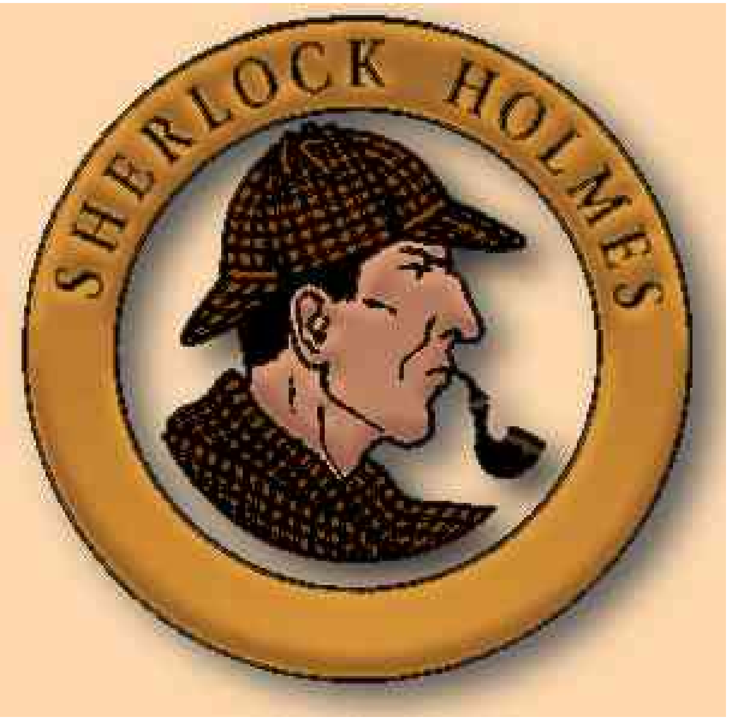
Tatatataaaaa: The nine rules

1. Understand the system
2. Make it fail
3. Quit thinking and look
4. Divide and conquer
5. Change one thing at a time
6. Keep an audit trail
7. Check the plug
8. Get a fresh view
9. If you don't fix it, it ain't fixed

Rule 1: Understand the System

- "It is not so impossible, however,
  that a man should possess all knowledge
  which is likely to be useful to him in his work,
  and this I have endeavoured
  in my case to do."

  Sherlock Holmes



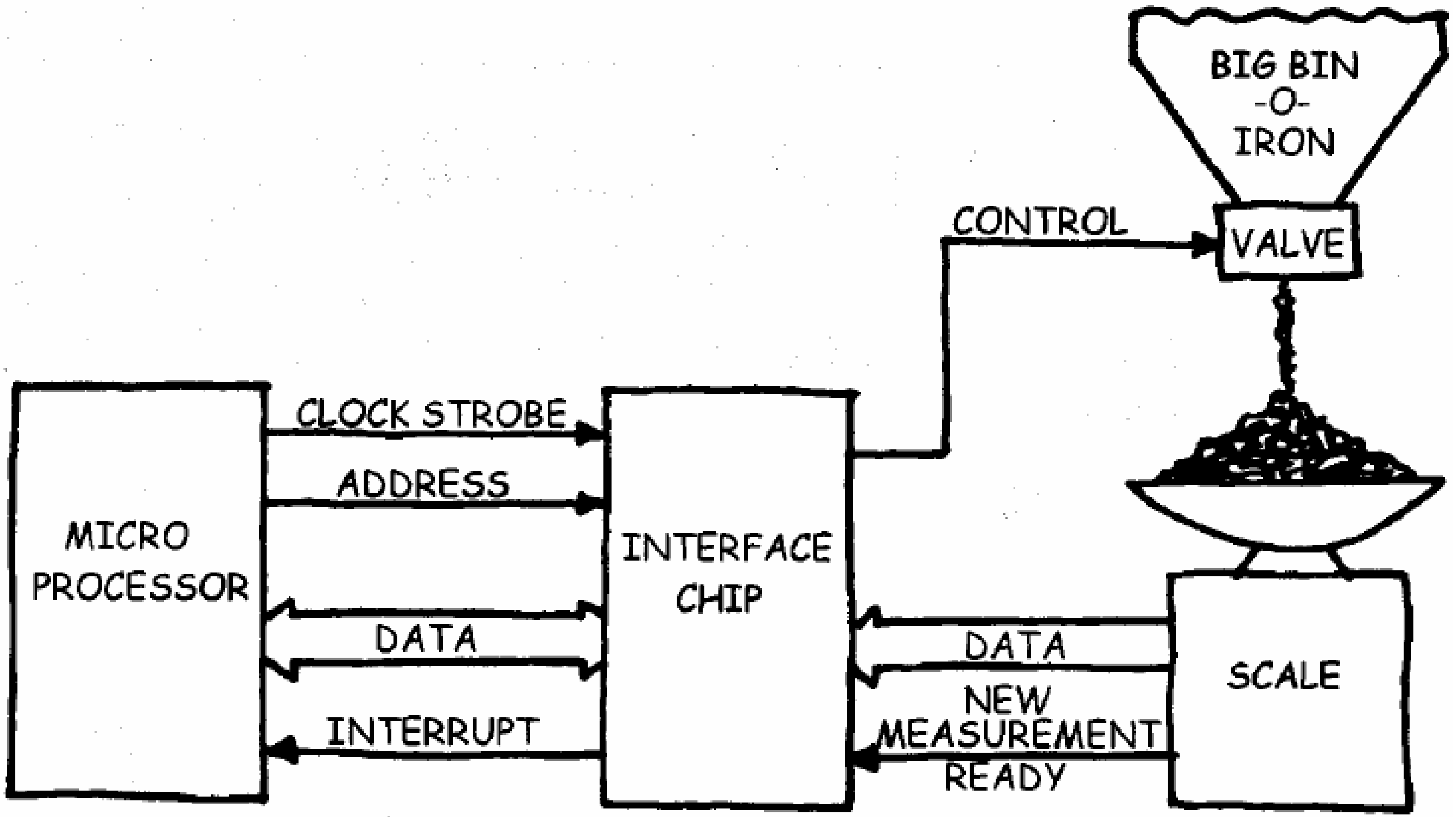
Understand the system: The war story (1)

- Situation: A microprocessor-based valve controller, built from a re-used design

- Problem: When the scale had a new measurement, the interface chip never passed on an interrupt to the processor
  - Even that was difficult to find out
  - No progress in finding out why

Understand the system: The war story (2)

* The system:



Understand the system: The war story (3)

- Critical event: A fellow engineer insisted that the designer reads the entire interface chip data book
  - page 37:
    "The chip will interrupt the microprocessor on the first deselected clock strobe"
  - The design never had deselected clock strobes:
    a few wires had been saved in the original design
    - as the original system had no interrupts at all

Understand the system: The essence

- You cannot find problems if you do not understand how the system is *supposed* to work
- Essentially, you have to "read the instructions"
  - Preferably *before* things go wrong

- Experience shows that the least understood parts of a complex system invariably have the most problems
  - You need the understanding at *design time*!

- Unfortunately, understanding a complex (software) infrastructure can be extremely time-consuming
  - This is usually the most time-consuming rule to follow
  - But also usually the most worthwhile

Understand the system: Read everything

- Subrule: Read everything, cover to cover
  - The finer the detail you miss, the more difficult the resulting problem

- War story:
  - 3-circuit boards had worked OK for some time
  - 4-circuit boards (otherwise identical) were introduced
  - They failed when they became hot. Nobody knew why.
  - Piece-by-piece comparison found different (but almost equivalent) types of memory chips to be the only difference
  - The design had accommodated all requirements of both types, except one: Wait time between read accesses.
  - The 4-circuit board chips required a longer time
  - (The 3-c. b. chips were also used beyond specification)

Understand the system: Know your basics

- Subrule: You need to understand the fundamentals of your technology
  - If you do not know what a strobe is, you cannot understand the data book
  - If you do not know what a low-endian word representation is, perfectly clean data may look garbled to you
  - If you specify UTF-8 without knowing what it is, you may never understand why your Umlaut characters do not show up as intended
  - If you understand only vaguely the purpose of methods you call, you will never find the defects in your program's logic
  - etc.

Understand the system: Know the landscape

- Subrule: You need to understand the purpose and interface of all major parts of your system and how they work together
  - Example: While riding your car, there is suddenly a regular tap-tap-tap sound.
  - Candidate parts: Engine, tires, fan
    - To make this list, you need to know your parts and their behavior
  - Diagnosis 1: It is getting faster as you go faster
    - So it is not the fan
  - Diagnosis 2: When you downshift, the tap-tap-tap stays the same
    - So it is not the engine, but the tires
    - To know this, you need to understand transmissions

The nine rules

1. Understand the system
2. Make it fail
3. Quit thinking and look
4. Divide and conquer
5. Change one thing at a time
6. Keep an audit trail
7. Check the plug
8. Get a fresh view
9. If you don't fix it, it ain't fixed

Rule 2: Make it fail

- "There is nothing like first-hand evidence."

  Sherlock Holmes

Make it fail: What and why

- For efficient debugging, you must be able to reproduce the failure at will
  - You need to find out the exact steps to make it fail

Why you need this:

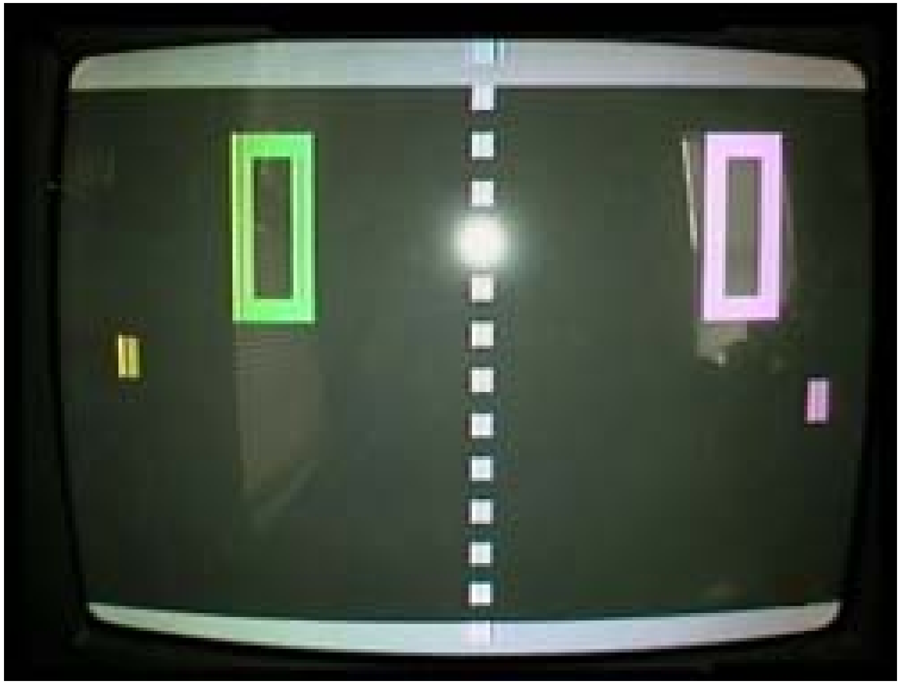
1. To look at the failure itself
   - and understand its characteristics
2. To be able to focus on finding the cause
   - rather than worry how to make it fail at all
3. So you can tell when you have fixed it

Make it fail: Practical hints

- Do it again
  - After you found out how to make it fail, do it once more to make sure you have the right procedure
- Start at the beginning
  - Make sure you can reproduce the failure from a repeatable (clean) initial state: restart everything from scratch
- Automate
  - If making it fail involves a lot of steps, automate their execution
    - see the war story below
  - If the conditions of the failure are rare, create them artificially
    - e.g. an allergy test by explicitly applying allergens
    - e.g. create heavy-load conditions to provoke a known heavy-load failure

Make it fail: War story (Automation)

- Context: Debugging an analog TV 'Pong' game
- Problem: The ball would sometimes wrongly bounce off the 'practice wall'
  - Manual playing kept attention away from observing the failure
- Automation:
  - Both ball position (x, y) and paddle position (y) were represented by voltages
  - Connect paddle y to the ball y voltage and the game will play itself



Make it fail: War story (Stimulation)

- Context: A house
- Problem: A particular window leaks in heavy rain -- but only sometimes
- Stimulation: Create artificial heavy rain by using a hose
  - The leaking happens only when the rain comes from southeast
  - Closely investigating the window finds a break in the caulking at that side of the window
  - After fixing the caulking, another hose test confirms that the defect has been fixed

Make it fail: What if it's intermittent?

- Some failures appear to be irreproducible (intermittent)

- But they aren't:
  - The factors evoking the failure are fixed
    - (remember?: laws of nature!)

- However
  - a. you may not know what the particular factors are
    - and there may be many to choose from or
  - b. you may not be able to control those factors
    - or the ones you would like to check for

Make it fail: What if it's intermittent? (2)

If you do not know the relevant factors:

- Try to find *some* relevant factor by trying to make the failure more frequent

- Method: trial-and-error experiments
  - Guess all kinds of conceivable factors
  - Change them and observe
  - If multiple factors are involved, only randomization helps
    - but randomize systematically

- Sometimes making it fail somewhat more often may be the best you can achieve
  - but that may be very helpful

Make it fail: What if it's still intermittent? (3)

If you cannot control the relevant factors
(or just still don't know them):

- Instrument the system to capture enough information about the few failures you get and about normal executions
  - and compare those two kinds
- Systematic differences usually provide the clue for finding and fixing the problem

- Problem: How can you make sure you fixed it?
  - You need to find a failure signature: Any run showing these conditions will fail
  - Then after your corrections, when you see such a run, but no failure, you know you have removed your problem.

Side note:
Never throw away a debugging tool

- You should keep useful ad-hoc debugging aids as if they were part of a product
  - Design them well, implement cleanly and document them
  - Store them in the version control archive etc.

- They may be useful again – even in unexpected ways

- War story:
  - When the TV 'Pong' game was shown to the investor, he asked for a "two practice walls mode"
  - Reason: *"For selling this thing, we need a demo mode to be run in the shop windows"*
  - Solution: The "robot player" developed as a debug aid

The nine rules

1. Understand the system
2. Make it fail
3. Quit thinking and look
4. Divide and conquer
5. Change one thing at a time
6. Keep an audit trail
7. Check the plug
8. Get a fresh view
9. If you don't fix it, it ain't fixed

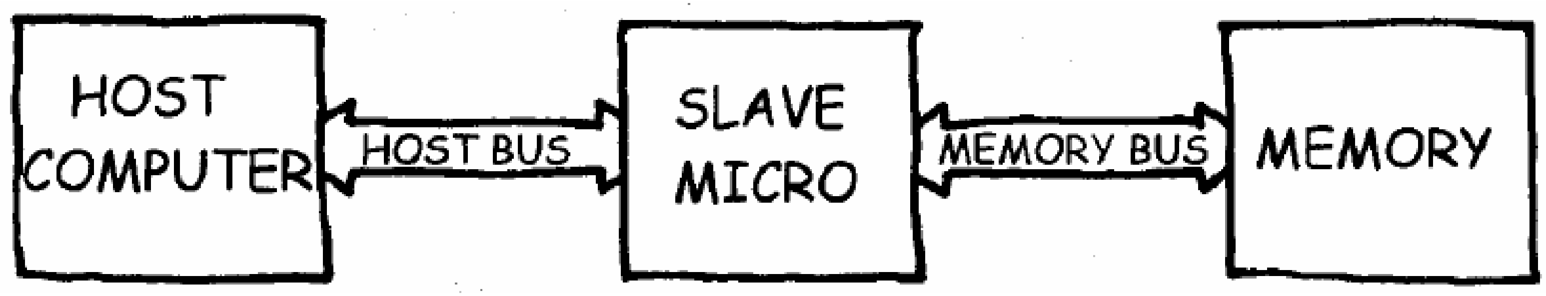
Rule 3: Quit thinking and look

- "It is a capital mistake to theorize before one has data. Insensibly one begins to twist facts to suit theories, instead of theories to suit facts."

  Sherlock Holmes

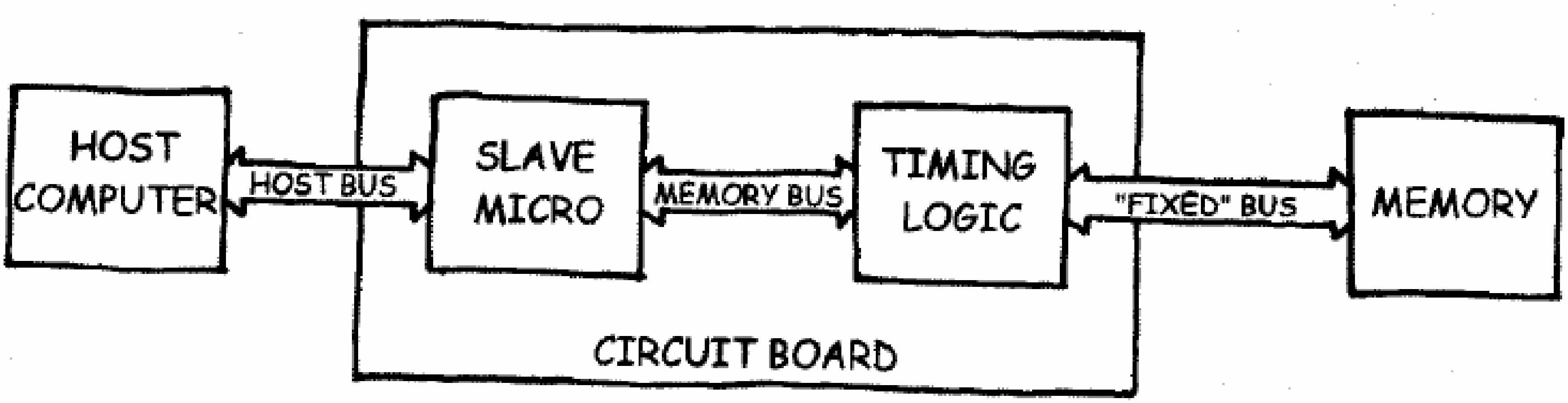
Quit thinking and look: War story

- PC card with slave microprocessor failed sometimes after the program upload
  - Memory checksum was incorrect
- Several junior engineers were assigned to fix this
- Their first test: Repeated writes into a register on the card microprocessor
  - Result was always correct



Quit thinking and look: War story (2)

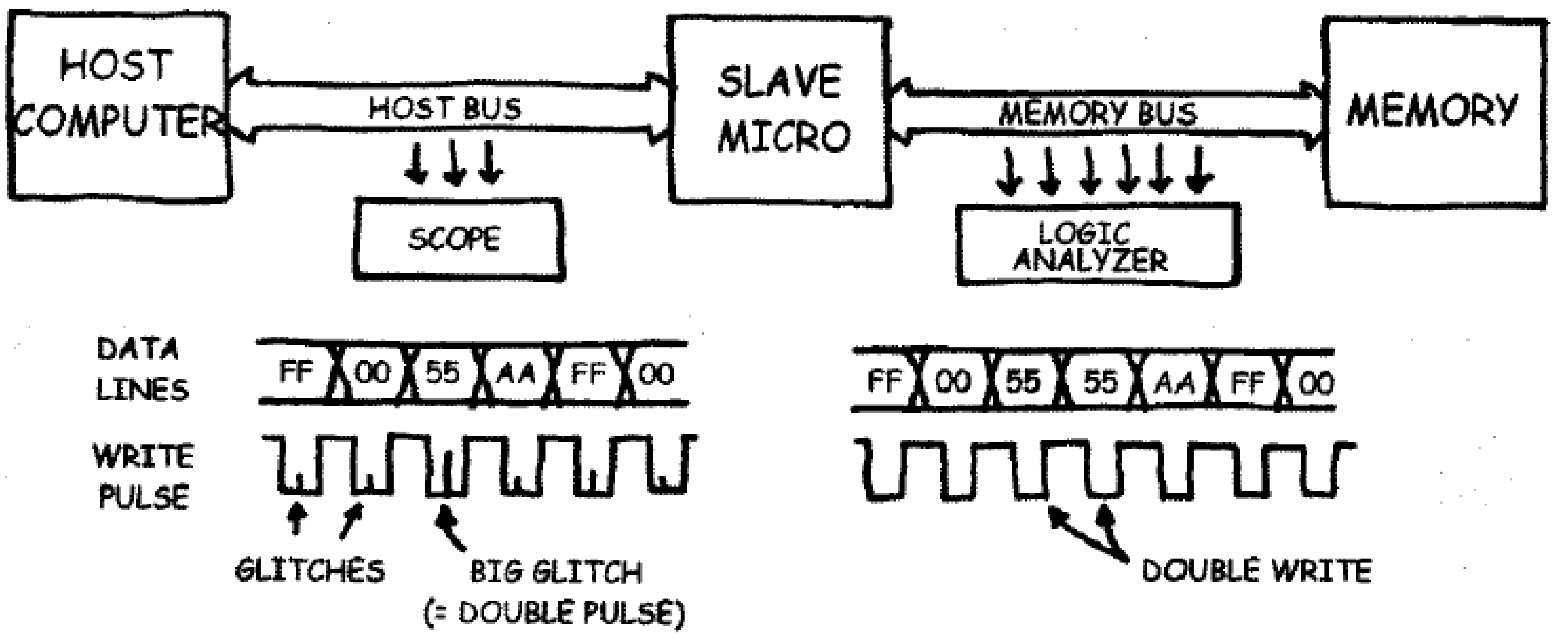
- Conclusion: Data transfer into the card works alright
- Next step: Understand the system
  - They analysed the memory interface circuits
  - They found that its timing design was borderline
- They assumed this was the problem
- They worked out an additional "fix-the-timing" circuit
  - That took several months!



Quit thinking and look: War story (3)

- Result: The card failed just as often as before
- Now a senior engineer stepped in and insisted they first see the actual failure
- He hooked up a logic analyzer to the memory bus and observed the results of repeated writes of the following pattern (to subsequent byte addresses): 00 55 AA FF
  - He sometimes found something like 00 55 55 AA FF
- So the writes to the card could be duplicate sometimes
  - He checked the write pulse to the card and found it to have noise which sometimes made it look like two pulses
- In the junior engineers "write register" test, this could not be observed
  - Writing the same value twice *to the same register* is not a problem

Quit thinking and look: War story (4)



Quit thinking and look: Subrules

- Subrule: See the failure
  - We tend to jump to conclusions when really what we are seeing is the consequence of a failure, not the failure itself
    - The junior engineers never saw the timing fail
- Subrule: See the details
  - Looking once is seldom enough
  - More typically, each looking provides a little more information; you will understand the failure bit by bit
- Subrule: Now you see it, now you don't
  - Seeing the actual low-level failure mechanism will be helpful later on when verifying a fix
- Subrule: Instrument the system
  - Looking from the outside may not be easy or good enough
  - Build observation aids (*instrumentation*) into the system

Quit thinking and look: Pump war story

- A person X did a favor to his neighbor P, who was a pump salesman
  - P: "Thanks. If ever you need a new well pump, I'll set you up with the best pump there is."
- While X is on a business trip, his wife W hears a strange new noise: a motor running for a few seconds every few hours.
- She calls P, he listens in and says: "It's the well pump".
- He calls his people and they replace the pump.
  - In the process, the well gets stirred and muddy
- The new pump works nicely (except for the mud)
  - **but the noise is still there as before.**
- Reason: X has left a compressor turned on in the garage

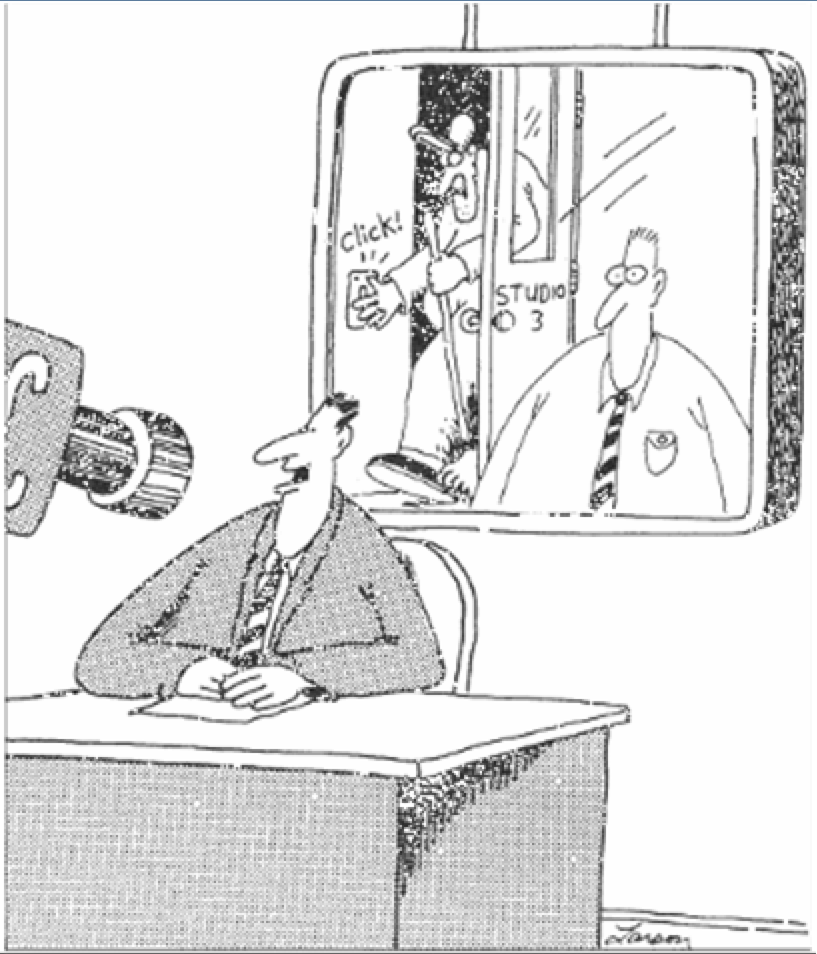
See the failure!

Quit thinking and look: Crash war story

- A server computer crashed late every night
  - Always at approximately the same time
- The administrators had restart logs, but no indication of the cause of the crashes
  - They figured it had to be "something automatic"
- They monitored processes for weeks, but found no correlations
- The one decided to stay late to actually observe the failure
  - and sure enough, shortly after 11pm, the computer failed: it lost its power supply
  - Reason: The janitor had pulled the plug, to use the outlet for the vacuum cleaner

See the failure!

Note: Janitor-related problems are fairly common!



"My next guest, on the monitor behind me, is an organized crime informant. To protect his identity, we've placed him in a darkened studio— so let's go to him now."

Quit thinking and look: Instrument the system

At design time:

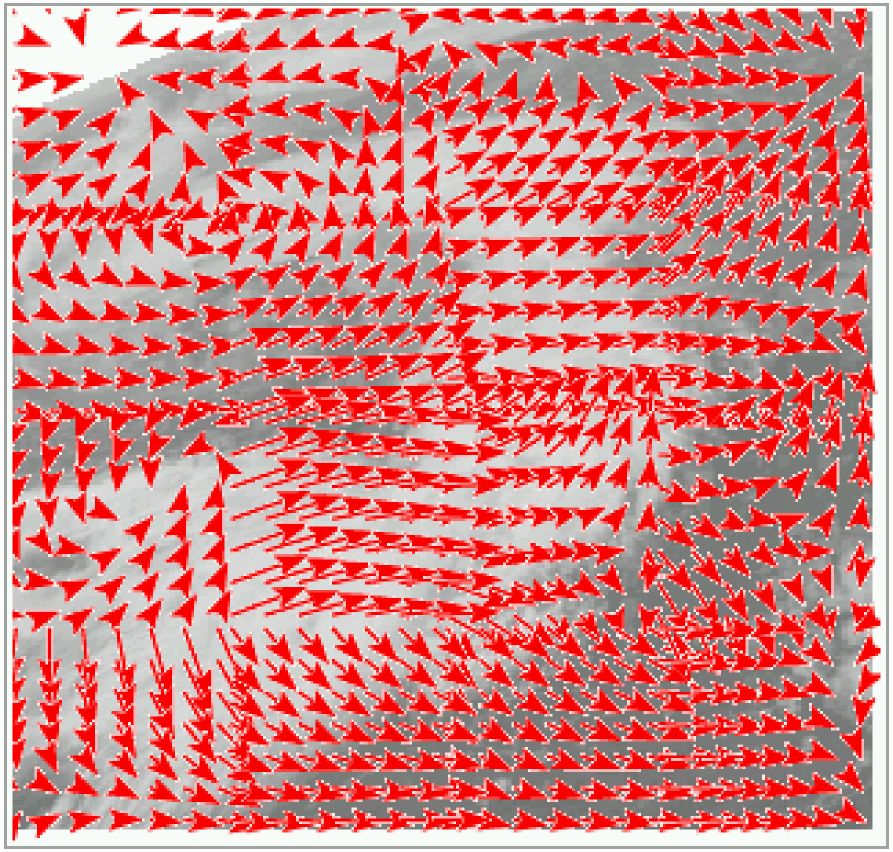
- In hardware (or embedded software) add plenty of monitor signals or even sensors or even displays
- In software, add plenty of tracing functionality
  - Preferably configurable at run time
  - Leave most of that in, even in production code
  - Use consistent, structured output formats for simple analysis
  - Always include time stamps

At debug time (ad-hoc instrumentation):

- Make sure the problem is still there after you built in your instrumentation
- Make sure it is still gone after you take it out (if any)

Quit thinking and look: War story

- A video compression software had surprisingly bad image quality on moving objects
- So the programmers analyzed the motion estimation
  - which encodes a moving object as an x/y translation of some part of the previous picture



Quit thinking and look: War story (2)

- They added code that would indicate motion estimation in the video by colored dots
  - Color for direction: down=orange, left=green etc.
  - Brightness for motion magnitude
  - The tests showed much fewer dots for left-right movements than for up-down movements

- They added output of all motion search results
  - and found that only few horizontal positions had matches

- There was a simple coding error in the horizontal search

See the details!

Quit thinking and look: Instrument the system (2)

- Sub-subrule: Don't be afraid to dive in
  - It is often advisable to do some preparation work (such as rebuilding the system with instrumentation put in) rather than try to debug "from the outside"
- Sub-subrule: Add instrumentation on
  - If your system is not instrumented (enough), you may be able to view some things at the existing network interfaces,
    - e.g. between application server and database server
- Instrumentation in daily life
  - Medicine: add-on (X-ray, ECG, EEG, etc.), built-in (marker genes etc.)
  - Plumbing: gauges (pressure, temperature, fill level), hydrogen sulfide added to natural gas

Quit thinking and look: Heisenberg principle

- The Heisenberg uncertainty principle:
  - When measuring location and momentum of a particle
  - it is impossible to measure both at once arbitrarily precisely
- Meaning (very roughly): It is a law of nature that instrumentation affects the observed system
- This principle often is at work in debugging, too (not literally, though)
  - Debuggers slow down the software and affect timing, change locations of code/data, neighbors, etc.
  - Instrumentation in the code increases code size and perhaps data size
  - Almost any change can influence instable hardware
- Hence, some problems disappear under instrumentation
  - They are often called 'Heisenbugs'

Quit thinking and look: Role of guessing

- The rule proposes looking to be better than guessing
- However, you often cannot get anywhere without guessing
- The rule is

  Guess only to focus the search
  - Positive example: the motion estimation war story
- After guessing you need to confirm your guess
  - Negative example: the pump war story

- There is one case where attempting a fix without confirming the guess may be the right idea:
  - The cause is highly likely and the fix is cheap
    - e.g. a burnt-out light bulb

The nine rules

1. Understand the system
2. Make it fail
3. Quit thinking and look
4. Divide and conquer
5. Change one thing at a time
6. Keep an audit trail
7. Check the plug
8. Get a fresh view
9. If you don't fix it, it ain't fixed

Thank you!