Avoiding the Top 43 Embedded Software Risks

Embedded Systems Conference SV
Updated: May 3, 2011

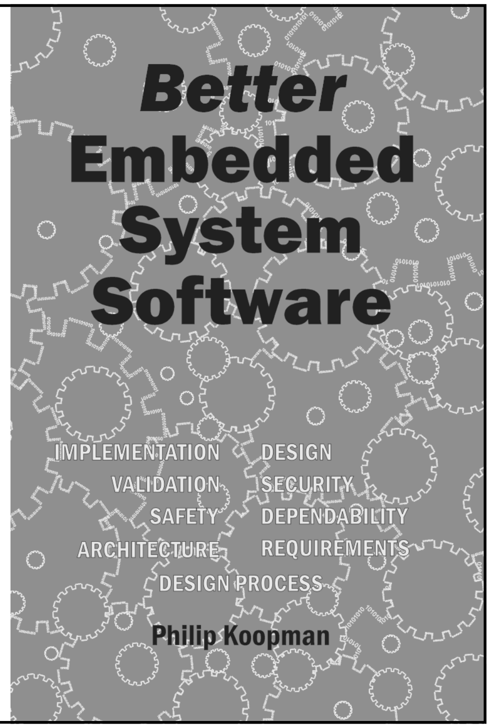
Philip Koopman
Carnegie Mellon University
http://BetterEmbSW.BlogSpot.com/

Overview

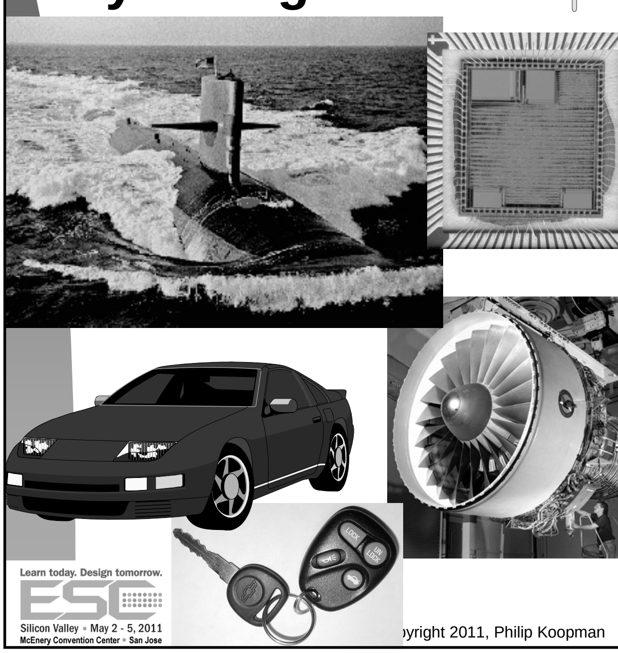
- How to mitigate embedded software risks
  - Data from 90+ design reviews spanning a decade
  - What teams got right and 43 areas they got wrong
- Best practice areas that can mitigate these risks
  - 17 general areas that address the risks
    - Specific practices that address all 43 areas
  - Most teams don't have resources to do them all
    - But most teams should be doing some
    - Which you should do depends upon your situation
    - Pick the low hanging fruit first to get best payoff

Talk Based on the Contents of My Book

- www.koopman.us
  - Discount and free international shipping
- Amazon.com
  - Geos Fulfillment is the publisher's direct sales channel



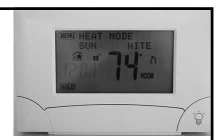
My Background



Types of Systems Surveyed

- Transportation
  - Automotive, train, navigation
- Chemical processing
  - Metering, flow control, analysis, automation
- Buildings
  - Heating/Ventilation/Cooling, building security, elevators
  - Lighting, electrical switching, domestic hot water
- Telecommunications and data centers
  - Climate control, power regulation, power switching, power backup, monitoring
- Underlying technology
  - Real time, safety, security, dependability
- Mostly excludes:
  - Consumer electronics, robotics, DSP

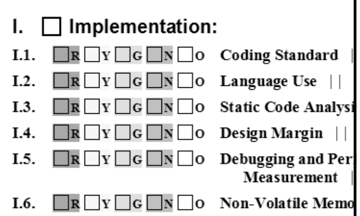
Developer Background



- No “typical” embedded developer, except what they are NOT
  - Almost no formally trained software engineers; few computer scientists
  - A distinct minority are formally trained computer engineers
- Most common development teams and environments:
  - Engineering domain experts: mechanical, electrical, automotive, HVAC, …
  - Smallish team sizes: 1 to 25 developers
  - Embedded languages: C, C++, assembly, a little Java; no custom ICs
  - Small to medium projects: 1000-1M lines of code
  - Medium size production runs: 1,000-20,000 units; Cost $20-$20K/unit
  - Old-school process models: Waterfall, Vee
  - Senior designers in US; common to have China, India team members
  - Small systems had no RTOS, bigger systems had one
- But, encountered at least one of almost everything
  - All-China team, all-Italy team, more/fewer units/year, Agile, …
  - And this advice will generally help all of them

Design Review Approach

- General approach:
  - Pre-visit review of available documents (if any)
  - On-site high level review of product
- Use a risk screening checklist to hunt for additional risks
  - Reviewer selected subset of 120+ questions based on pre-review (full list is proprietary)
  - Graded as “red” / “yellow” / “green”
  - (Some reviews didn’t use checklist, so we did after-the-fact binning)
- What we care about: “Red” Issues



Study Methodology

- Retrospective of review reports (10+ years; 90+ reviews)
- Tallied risk list bins in reports
  - In some cases mapped ad hoc description to bins
- Results:
  - A list of 43 distinct red flag bins
- “Red Flag” means “don’t ship until you fix this”
  - Not simply “you should do this because it is best practice”… … but rather “this will cause a big problem for *this project*”

Technical Risks

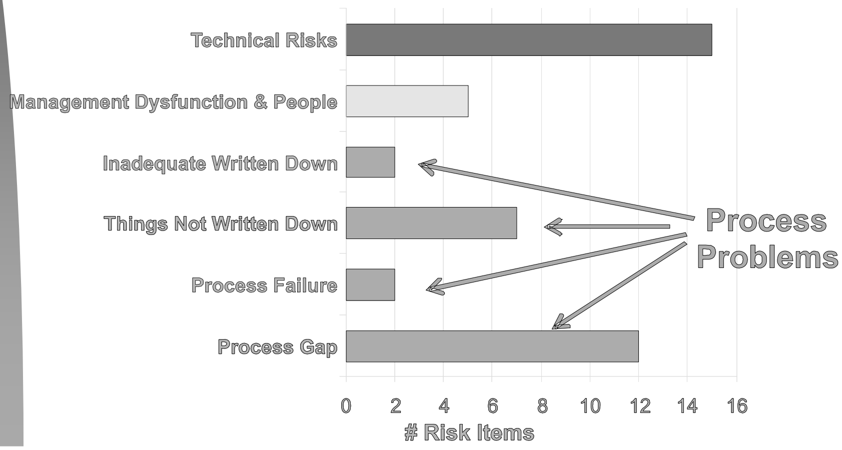
- Most developers were domain experts, not computer experts
  - Usually a senior developer who had learned the hard way
  - Generally capable engineers … self-taught from books/eval kits
- I expected to find lots of technical issues
  - There were some, but … not that many rookie technical mistakes
  - Mostly problems with *complexity* or *advanced embedded topics*
- In general, technical problems:
  - Corresponded with common holes in intro embedded textbooks
  - Mostly were things that were hard to find in simple testing
    - In other words, most projects got the basic functionality right
  - The problem areas tended to be things they didn't do (lack of time; lack of knowledge)

The 43 Risk Areas

1. Informal development process
2. Not enough paper
3. No written requirements
4. Requirements omit extra-functional aspects
5. Requirements with poor measurability
6. No defined software architecture
7. Poor code modularity
8. Too many global variables
9. No message dictionary for embedded network
10. Design skipped or created after code is written
11. Flowcharts are used in place of statecharts
12. Inconsistent coding style
13. Ignoring compiler warnings
14. No peer reviews
15. No real time schedule analysis
16. Use of home-made RTOS
17. Inadequate concurrency management
18. No methodical approach to user interface design
19. No test plan
20. No stress testing
21. No defect tracking
22. No run-time fault instrumentation nor error logs
23. Defect resolution for 3rd party software
24. Disaster recovery not tested
25. Insufficient consideration of reliability/availability
26. Insufficient consideration of safety
27. Insufficient consideration of security
28. No IP protection plan
29. No or incorrect use of watchdog timers
30. Inadequate system reset approach
31. High requirements churn
32. No version control
33. No backward compatibility plan
34. No software update plan
35. Lessons learned not being recorded
36. Acting as if software is free
37. Use of cheap tools instead of good ones
38. High turnover and developer overload
39. No training for managing outsource relationships
40. Resources too full
41. Too much assembly language
42. Project schedule not taken seriously
43. No Software Quality Assurance (SQA) function

What Is The Big Picture?

- Most problems are with process *omissions*
  - *But, we still have technical areas to talk about too!*



17 Good Practice Areas

1. Define your development process
2. Write good requirements
3. Use a good architecture
4. Create a written design
5. Use good coding style
6. Use peer reviews
7. Use real time analysis
8. Manage concurrency
9. Design a user interface
10. Follow a test plan
11. Manage issues/defects
12. Design for quality attributes
13. Use watchdog timer correctly
14. Manage change
15. Don't think software is free
16. Have slack resources
17. Make sure you follow your process

A Tour Of Good Practices

- Remember, you don't have to do all of these
  - But, you should harvest the low hanging fruit
- Some of this sounds like "software engineering"
  - … but really it is just "good engineering"
  - It's about *why* you do things, not just about paperwork
- Knowing how to solder doesn't make you a hardware engineer
- Knowing how to write lines of code doesn't make you a software engineer
- Knowing how to solder *and* write lines of code doesn't make you an embedded systems engineer

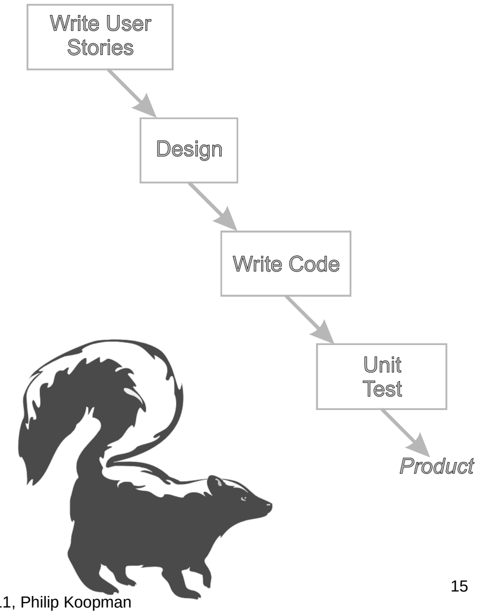
Define Your Development Process

(Risk #1: Informal development process)

- Development process is a set of steps, e.g.,
  - Define Requirements
  - Write Code
  - Acceptance Test
  - Ship
- If the steps aren't well defined, you don't have a roadmap
  - (If you don't really have one, get some help to define one!)

Is This A Well Defined Process?

- Any missing pieces?
- How do we know what the design is?
- How do we know the product is ready to ship?
- If this were a hardware block diagram, what would be missing?



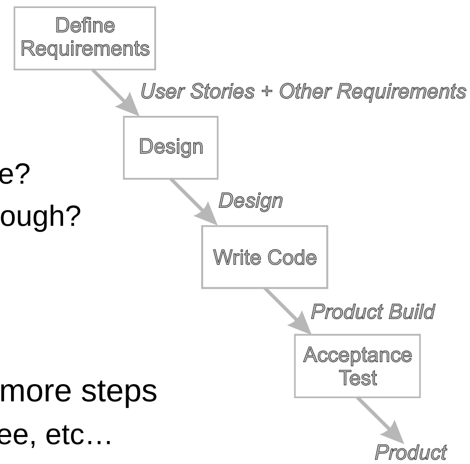
A Good Development Plan Has:

- Development steps
  - Activities inside process boxes
- Defined output from each step
  - Paper, code, etc. – what are the work products?
    - Artifacts” in software-engineer speak
- A risk management approach
  - Exception handling, actual “management” of process
- A way to measure success
  - Is the product good enough to sell?

If it isn’t written down, it didn’t happen

A Better Process Example

- ☑ Activities
- ☑ Work Products
- ☸ Risk management
  - Where is that in this picture?
  - Is final acceptance test enough?
- ☑ A way to measure success
  - "Passes acceptance test"



- Process usually has many more steps
  - Can be Agile, Waterfall, Vee, etc…
  - But has to be *defined* including both processes (boxes) and artifacts (arrows)

Using The Right Amount of Paper

(#2 Not enough paper)

- Use the right amount of paperwork (not zero)
  - Be clever in minimizing paperwork

- Product document package should include *at least*:
  - Development approach (the development plan)
  - Requirements
  - Architecture
  - Design
  - Test plan & test results
  - Implementation
  - Reviews
  - Maintenance

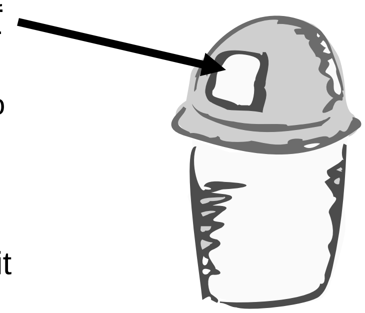
Keeping "Paper" Light

- If it isn't written down it didn't happen…
  - … but it doesn't have to be a 1000 page novel!
- Make use of:
  - Spreadsheets
  - Fill-in-the-blank templates
  - Powerpoint
  - Photos of whiteboards + notes
- The most effective paperwork:
  - Fits on a single "sheet"
  - Can be found via searching
  - Provides useful value … so it actually gets made

Learn today. Design tomorrow.
ESC
Silicon Valley • May 2 - 5, 2011
McEnery Convention Center • San Jose

Modest Proposals For Paperwork

- Every development step should produce "paper"
  - Every process arc has paper in defined format
  - Make it the simplest paper you can justify
  - But, zero paper is not acceptable
- If paper gets out of date, throw it *and* the associated code away – *right now*
  - If it's not important enough to do well, why are you doing it at all?
- If you decide to skip paper, throw the project away when the developer stops working on it



Learn today. Design tomorrow.
ESC
Silicon Valley • May 2 - 5, 2011
McEnery Convention Center • San Jose

“But, We Don’t Need Paper”

- Really great software has been created without paper
  - Works best if *all* your developers are well above average
  - And nobody ever changes jobs, taking knowledge with them
  - *But that just doesn’t scale*

- Five Forebodes Failure
  - Teams with exactly 5 developers often failed
    - Usually previous project had 3 or 4
    - Teams of 6 or more had heavier process
  - My conclusion: with 5 people you need “paper”
    - Max 4 people can informally coordinate (neighbors)
    - Larger projects have more coordination overhead
    - Much higher risk if you use an ad hoc process for >4 people
    - Paper for fewer than 5 still helps

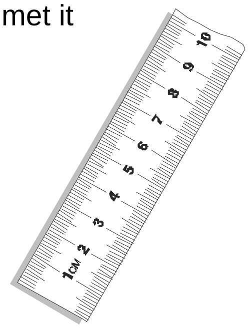
Write Good Requirements

(#3 No written requirements) -- User Stories are OK
(#4 Requirements omit extra-functional aspects)
  - You can’t keep things straight without having written requirements
  - Saying “just like last system except” is a problem too
- Rigorously written
  - Precise: “X shall do Y” or “supports following sequence of operations”
  - Unambiguous: good technical writing practices
  - Describes “what” rather than “how” – it’s not a design
- Traceable: how do you make sure you met it
  - E.g., each one has a number that traces to acceptance tests
- Covers:
  - What the system should do
  - What the system should *not* do
  - Extra-functional aspects (security, safety, dependability, performance)
  - Standards, constraints, certifications

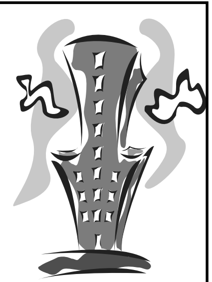
Making Requirements Measurable

(#5 Requirements with poor measurability)

- Requirements should also be measurable
  - If you can’t measure it, you can’t know you met it
  - Beware of subjectivity, e.g., “User Friendly”
- Don’t require perfection
  - You can’t get it … and you can’t measure it
- If in doubt, write a test metric with the requirement
  - “Never crashes” ➔ “Does not crash in 1 week of stress testing”
- Collect field data with a flight recorder to confirm outcomes



Use A Good Architecture

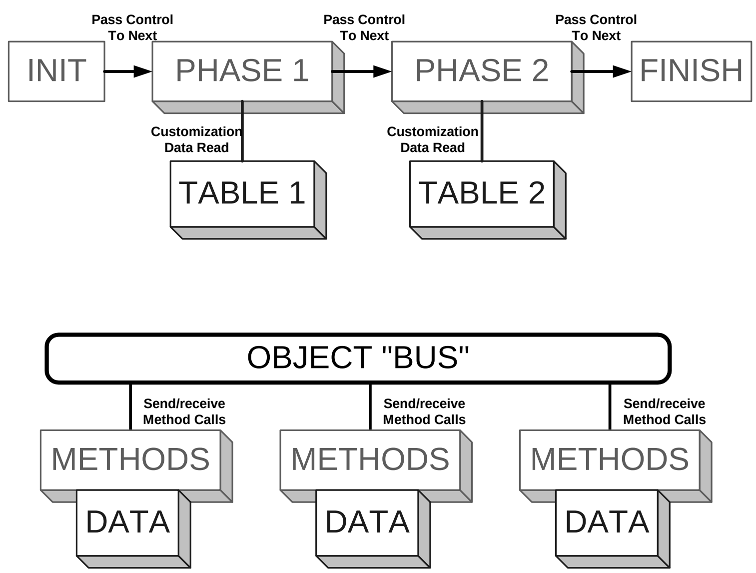


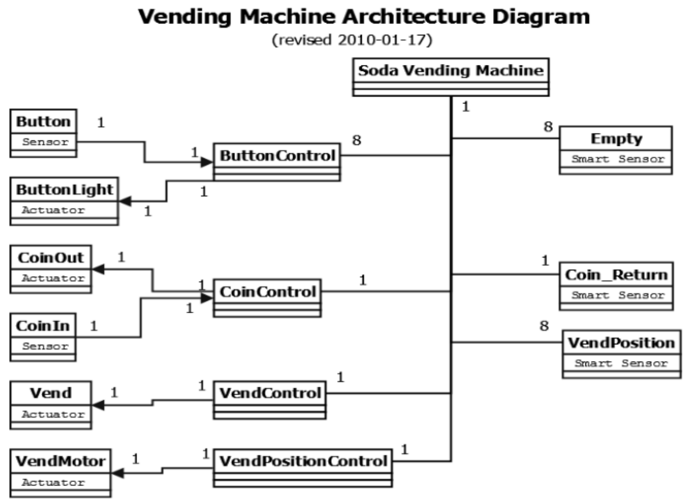
(#6 No defined software architecture)

- Would you build a house without a floor plan?
  - (If you did, how would it turn out?)
- Would you build a computer without a block diagram
  - (If you did, how would it turn out?)
- So why do we think it is OK to just write code without an architecture?
  - The IT guys always have a SW architecture diagram
    - Are we so smart we don’t need one?
    - Or are our systems so trivial it isn’t worth the bother?

The Basics of Software Architecture

- Create a “boxes-and-arrows” diagram
  - Boxes are objects or activities
  - Arrows are flows (data, control, …)
- Need to be able to say:
  “Here is a picture of my high level software organization.”
- Helpful guidelines (similar to HW block diagrams)
  - Every box and arrow has a defined meaning
  - Fits legibly one on letter size sheet of paper
  - Can be hierarchically nested to multiple sheets
  - Can have more than one type for the system
    - Call graph, data flow diagram, class diagram, etc.





Black line = "is comprised of"
Blue line = "analog connection"
Numbers are replication counts

Black box = SW function
Blue box = I/O hardware

Global Variables Are Evil

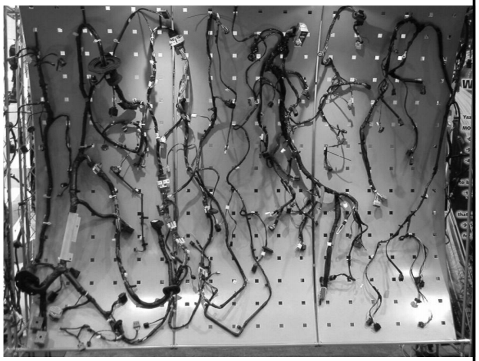
(#7 Poor code modularity)
(#8 Too many global variables)

- Good architectures are modular
  - Low coupling
    (different parts are unrelated)
  - High cohesion
    (each part is homogeneous)
  - Meaningful levels of decomposition and abstraction
- Global variables are shared across modules
  - Minimize using them (use local variables when possible)
    - If you are using them because you have insufficient RAM, see discussion on "software isn't free" later
  - If you must use them:
    - Ensure only one place each is written
    - Limit visibility to a single module ("static" keyword)
    - Try to keep them together so they are easy to find

Embedded Network Architecture

(#9 No message dictionary for embedded network)

- Always have a message dictionary
  - All message types
  - Header and other info
  - Data meaning and format
  - Sender/receivers, period, deadline, etc.
  - Globally visible network variables, if applicable
- If you must use a custom protocol, document it
  - What happens if the one guy who knows the protocol wins the lottery and retires?



Example CAN Message Dictionary

Sender Node Name	Message Name	Deadline (ms)	Message ID	Source Node Type	Replication Type	Base CAN ID
VendPosition Sensor	mVendPosition	50	100	10	s	0x08640A00
VendControl	mVend	50	200	20	none	0x08C81400
VendPositionControl	mVendMotor	50	300	30	none	0x092C1E00
ButtonControl	mButton	100	400	40	s	0x09902800
CoinReturn Sensor	mCoinReturn	100	500	50	none	0x09F43200
CoinControl	mCoinCount	100	600	60	none	0x0A583C00
Empty Sensor	mEmpty	500	700	70	none	0x0ABC4600

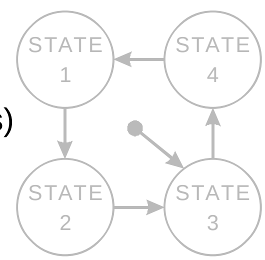
Create A Written Design

(#10 Design skipped or is created after code is written)

- Would you design an engine with no drawings?
  - Would you lay out a circuit board with no schematic?
  - Would you write lines of code with no design?
- A design lets you think at a high level
  - Concentrate on overall flow – not coding details
  - Get reviews more efficiently

- *Self-documenting code isn't*
  - Designs extracted from the code are a waste of time
  - JavaDoc documents code, but is not a *design*

Always Use Some Statecharts

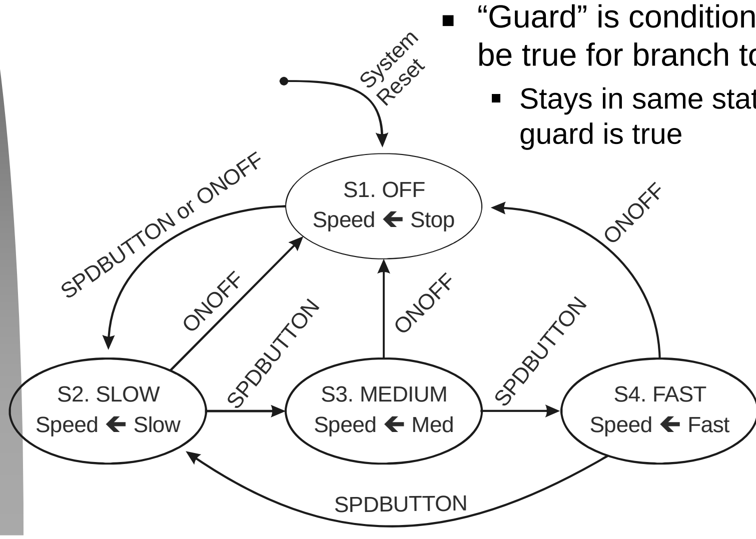


(#11 Flowcharts used in place of statecharts)

- Flowcharts can help with design, but…
- Most embedded systems are state based
  - States represent operating modes (idle, run, ramp-up, ramp-down)
  - States represent display modes (think digital watch)
  - States create model of external environment
- Flowcharts are OK for memory-less control flow
  - If you have duplicated "if" conditions, statechart might be better
  - Psuedocode is too loose – not good in practice most times
- Model based design can help, but is not a magic wand

Statechart Example

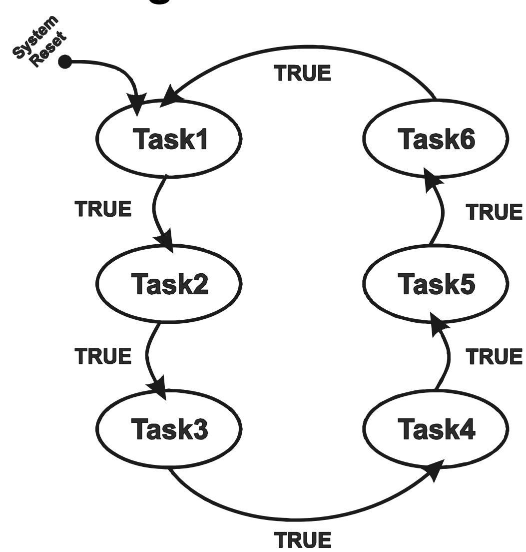
- "Guard" is condition that must be true for branch to be taken
  - Stays in same state if no guard is true



Switch-Based Statechart Code

```
enum CurrState
{OFF, SLOW, MEDIUM, FAST}; // define states

#define SpdOff  0   // define speed constant values
#define SpdSlow 10
#define SpdMed  15
#define SpdFast 25

CurrState = OFF;  // initialize state machine to OFF

while (1) // do forever
{
  switch (CurrState) {
  case OFF:    // State S1
    speed(SpdOff);         // Take action in state

    // Test arc guards and take transitions
    if (SpdButton() == TRUE || OnOffButton() == TRUE)
    {CurrState = SLOW;}
    break;   // go to end of switch statement
```

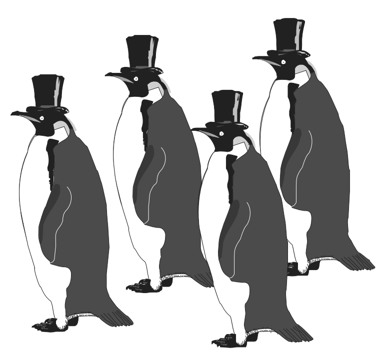
What's Wrong With This Statechart?



Use Good Coding Style

(#12 Inconsistent coding style)

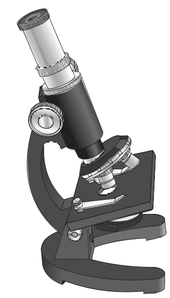
- Everyone has their favorite coding style
  - It doesn't matter (much) which style you use
  - But have everyone use the *same* *defined* style
- Include things such as:
  - Title block contents
  - Commenting guidelines
  - Assertions
  - Language usage rules
  - Naming conventions



Static Analysis & Warnings

(#13 Ignoring compiler warnings)

- Use static checking to keep your code clean
  - It's like getting a free automated (partial) design review
  - Compiler warnings tell you something is fishy
    - Language definition ambiguities
    - Risky language use
    - Common mistakes
  - Code should compile with *no warnings*
- Some embedded compilers give poor warnings
  - Try a higher-end compiler
  - Try using splint (a "lint" tool that does static checking)



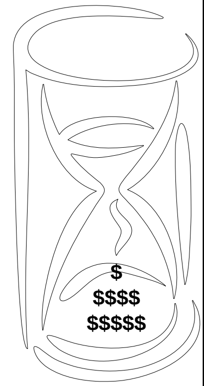
Example Warnings

- if (a=b) { …. Do something… }
- // feet & meters are int typedefs
  feet a;   meters b;
  b = a;
- Uninitialized variable
- Unreachable code
- Failure to conform to a language subset
  - E.g., Misra C language subset for safety critical SW

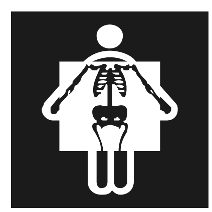
Use Peer Reviews

(#14 No peer reviews)

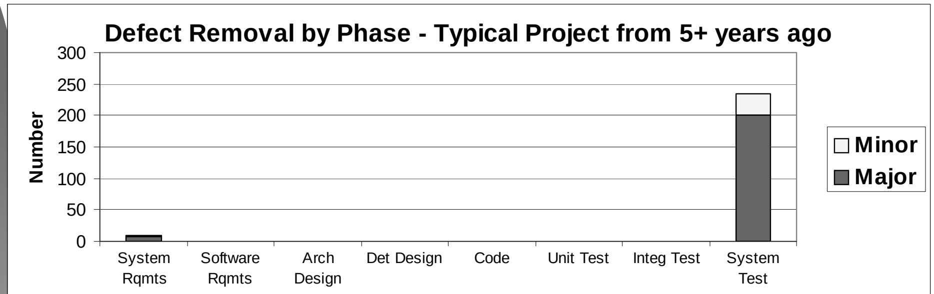
- Peer reviews are the most cost effective way to find bugs
- *Good* embedded coding rate is 1-2 lines of code/person-hr
  - (Across entire project, including reqts, test, etc.)
- How much does peer review cost?
  - 4 people * 100-200 lines of code reviewed per hour
  - Say 300 lines; 4 people; 2 hrs review + 1 hr prep
    = 25 lines of code reviewed / person-hr
  - Reviews are only about 5%-10% of your project cost
- Good peer reviews find about *half the bugs!*
  - *And they find them early, so cost to fix is lower*

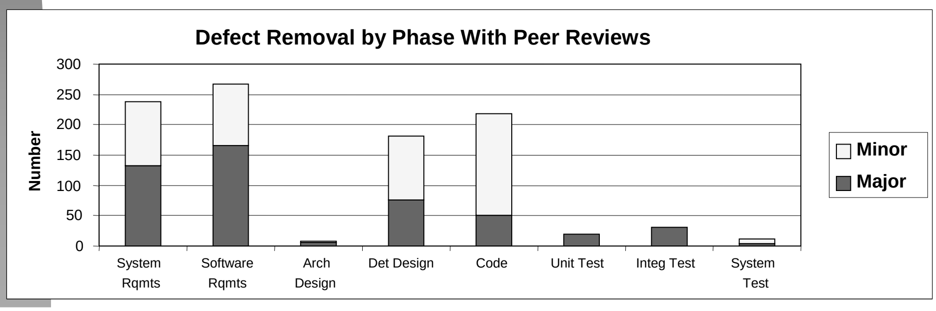


What Should You Review?



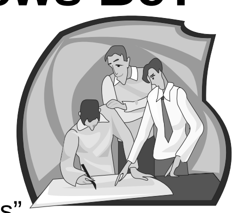
- Review everything that is in writing
  - (From earlier, every project activity should produce a written artifact)
  - Early reviews have higher bang-for-buck
    - Review requirements and designs
    - Don't wait until you are at code to start reviews
    - Most reviews happen before testing, so possible to reduce total cost of bugs dramatically with reviews
- Things you can review:
  - Requirements, architecture, design, implementation, test plan, user guide, schedule, development plan, real time schedule, …





[Source: Roger G., Aug. 2005]

How Formal Should Reviews Be?



- The more formal the review, the higher the payoff
  - Formal reviews take more effort; but are far more productive
    - We mean use these: “Fagan style inspections”
  - Formal reviews of absolutely everything should still be less than perhaps 10% of total project cost
    - In return, you find half of your bugs much earlier
- Informal reviews are better than nothing
  - Pair programming, shoulder surfing, e-mail pass-arounds are better than nothing
  - Payback for on-line review tools is a question mark
    - Reduces social interaction, training of junior developers

Rules For Good Reviews

1. Inspect the item, not the author
2. Don't get defensive
3. Find problems – but don't fix them in the meeting
4. Limit meetings to two hours
5. Keep a reasonable pace
   - 150-200 lines per hour
6. Avoid "religious" debates on style

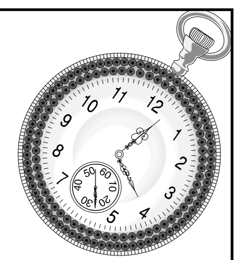
- Inspect, early, often, and as formally as you can
  - Use inspections (formal reviews) as much as possible

Peer Review Metrics

- Want to balance peer review with other efforts
- How do you know peer reviews are working?
  - Track that 40%-60% of defects are found by reviews
  - BUT, what if entering into Bugzilla is too expensive?
- Lightweight alternative:
  - Use a simple spreadsheet to record review results
  - Tally # of defects found and just aggregate numbers
  - Only enter in Bugzilla if defect is uncorrected after completing develop/peer review/bug fix cycle
- If reviews find < 40% of defects, reviews are probably broken

	Peer Review Template for Project X	
Date:	4/17/2011	
Artifact:	Xyzzy.cpp Functions: Foo(), Bar(), Baz()	
Reviewers:	Stella K., Joe B., Sam Q., Trish R.	
Size:	357	SLOC
Time Spent:	112	Minutes
# Issues:	3	
Outcome:	Re-Review of Bug Fixes Required	
Issue#	**Issue Description**	**Status**
1	Issue 1…..	Fixed
2	Issue 2….	Bugzilla
3	Issue 3….	Bugzilla
4	Issue 4….	Not a Bug
5		
6		
7		
8		
Status Key:	Fixed (trivial fix by author; no need to enter in defect list)	
	Bugzilla (entered into project defect system)	
	Not a Bug (false alarm)	

Use Real Time Analysis



(#15 No real time schedule analysis)

- If you need to meet real time deadlines, you need to do a formal real time analysis
  - List tasks, deadlines, periods, compute times
  - Use a well understood scheduling theory
    - Understand assumptions and limitations
    - If you do something ad hoc, eventually you'll be burned
- Use the simplest scheduling technique you can
  - Cyclic executive works great
    - Interrupts are tasks and need to be accounted for
  - If you use preemptive non-ISR tasks, use Rate Monotonic Scheduling
    - Don't use earliest deadline first

Rate Monotonic Scheduling 101

- Assume:
  - All tasks are periodic; Period = Deadline
  - Worst case compute time known for each task
  - All tasks are independent (no mutexes)
  - Task switching has zero latency and cost
  - Task periods are harmonic multiples (permits 100% CPU use)
- To schedule:
  - Assign priorities based on period (fastest = highest priority
  - If CPU utilization is less than 100%, it will work
    - The 100% limit is due to harmonic multiple periods
- If you need to violate assumptions, read up on this topic
  - It is easy to get things “almost” right ➔ ➔ ➔ wrong

Example Rate Monotonic Schedule

Task #	Period (P_i)	Compute (C_i)
T1	5	1
T2	**<u>16</u>**	2
T3	**<u>6</u>**	2
T4	60	3
T5	30	4

Task #	Priority	Utilization μ
T1	1	1/5 = 0.200
T3	2	2/<u>6</u> = 0.333
T2	3	2/<u>16</u> = 0.125
T5	4	4/30 = 0.133
T4	5	3/60 = .05
	TOTAL:	**<u>0.841</u>**

$$\mu = \sum \frac{c_i}{p_i} \leq N(\sqrt[N]{2} - 1) \qquad ; \mathrm{N} = 5$$

$$\mu = 0.841 \quad (not \leq) \quad 0.743$$

Not Schedulable!

Example Harmonic Rate Monotonic Schedule

Task #	Period (P_i)	Compute (C_i)
T1	5	1
T2	**15**	2
T3	**5**	2
T4	60	3
T5	30	4

Task #	Priority	Utilization μ
T1	1	1/5 = 0.200
T3	2	2/**5** = 0.400
T2	3	2/**15** = 0.133
T5	4	4/30 = 0.133
T4	5	3/60 = .05
	TOTAL:	**0.916**

$$\mu = \sum \frac{c_i}{p_i} \leq 1 \text{ ; Harmonic } P_i\{5, 15, 30, 60\}$$

$$\mu = 0.916 \quad \leq \quad 1$$

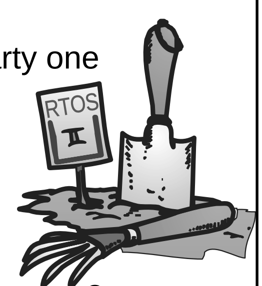
Schedulable, even though usage is higher!

Learn today. Design tomorrow. ESC Silicon Valley • May 2 - 5, 2011 McEnery Convention Center • San Jose

Don’t Use A Home Grown RTOS

(#16 Use of home-made RTOS)

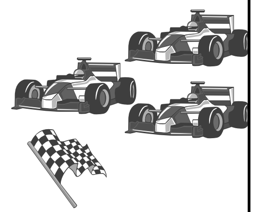
- If you need a preemptive RTOS, use 3rd party one
  - Getting an RTOS right is really, really hard
  - Even if you can get it right, it is a lot of work
  - Even if you do get it right, what happens in 10 years when you aren’t maintaining it?



- Ask yourself: is RTOS writing a core competency?
  - Shouldn’t you be spending that time on your products?
    - (See “software is free” later in this talk)
  - It’s not hard to find a mostly free RTOS these days
    - But it might be more cost effective to pay for one!

Learn today. Design tomorrow. ESC Silicon Valley • May 2 - 5, 2011 McEnery Convention Center • San Jose

Manage Concurrency



(#17 Inadequate concurrency management)

- Race conditions and data sharing problems
  - Tough to reproduce; tough to pin down
  - Very difficult to find and fix
  - You probably won't find them in normal testing
    - Look up "Therac 25"
- Consider concurrency for every shared variable
  - Use a mutex if you have to (see next slide)
  - Use something easier if you can (e.g., Fifo; mask interrupts)
  - Use standard approaches
    - You aren't good enough to invent a new approach (and neither am I)
  - Realize that this breaks scheduling independence assumption
    - Look up "Mars Priority Inversion"

Learn today. Design tomorrow. ESC Silicon Valley • May 2 - 5, 2011 McEnery Convention Center • San Jose

Example Mutex ("Mutual Exclusion")

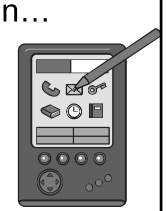
```
Mystruct Foo;       // Foo is shared by multiple tasks
volatile uint8 FooMutex = 0;  // 0 is nobody using
                              // 1 is in use (locked)

… somewhere in a task …
   uint8 InitialValue;  // Use "Test-and-Set" approach
   do {     SEI();      // Mask Interrupts
            InitialValue = FooMutex; // Save old value
            FooMutex = 1;            // Attempt to lock
            CLI();      // Unmask Interrupts
      } while (InitialValue != 0);  // Try until 0

  Foo.a = <newval>;     // We own Foo; make changes
  Foo.zz = <newval>;
  FooMutex = 0;         // Done with Foo; unlock it
```

Learn today. Design tomorrow. ESC Silicon Valley • May 2 - 5, 2011 McEnery Convention Center • San Jose

Design A User Interface

(#18 No methodical approach to user interface design)

- Most engineers are terrible at user interface design…
  … because most engineers aren't "normal"
  - And most engineering depts. aren't that diverse



- Do "user testing" where real users try things out
  - There are people who do user interaction for a living!
  - User interface principles: consistent, simple, user-centered
- Take into account use demographics & diverse use cases
  - Color-blind, arthritis, left-handed, hearing impaired, age
  - Polarized sun glasses, gloves, ear plugs
  - Internationalization, time zones, daylight savings time
  - A user interface checklist with the above can help

Learn today. Design tomorrow.
ESC Silicon Valley • May 2 - 5, 2011
McEnery Convention Center • San Jose

Follow A Test Plan

(#19 No test plan)
(#20 No stress testing)

- Key to testing is *coverage*
  - Each type of test has different coverage
- Unit test – might use code coverage
  - Did every line of code get exercised?
- Integration test – test component interfaces
  - Did every method and option flag get exercised?
- Acceptance test – traces to requirements
  - Did every requirement of system get checked?
- Test early to find bugs while they are cheap to fix
  - Usually: unit test, subsystem test, integration test, stress test, acceptance test, beta test

Learn today. Design tomorrow.
ESC Silicon Valley • May 2 - 5, 2011
McEnery Convention Center • San Jose

Written Test Plan

- Best approach is a written test plan
  - Usually this is a spreadsheet for embedded systems
  - For each test:
    - Traceability of test (e.g., which requirement)
    - Initial conditions
    - Test procedure
    - Expected result
    - Actual result and pass/fail

- Plan specifies desired coverage
  - Often can be a spreadsheet – one row per test
  - For each type of testing, how thorough should it be?
  - Bug prioritization
  - How you know you are done testing

Typical Coverage Strategies

- Unit Test (developers)
  - Fraction of lines of code executed (e.g., 92%)
- Peer Review (developers)
  - Fraction of lines new/modified code reviewed
- Subsystem test (testers+developers)
  - Fraction of modules exercised
- Integration test (testers)
  - Fraction of interfaces exercised
- Acceptance test (testers)
  - Fraction of system requirements exercised

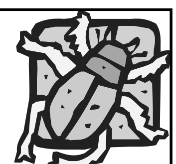
How Much Test Is Enough

- Get a reasonably good level of coverage
  - But, how much does test and other QA cost?

- For embedded systems, probably 50%-65% of total system cost(!)
  - Tester : Developer Ratios
    Web Apps: 1 : 5
    OK IT Code: 1 : 1
    Safety Critical Code: 5 : 1
  - If it really has to work, you need perhaps **<u>2 : 1</u>**
  - Embedded projects with marginal quality often at 1 : 1

- The good news: all verification/validation counts
  - Unit test, peer reviews --- all count as “test”!
  - So does other testing (and probably SQA)

<u>Manage Issues and Defects</u>



(#21 No defect tracking)

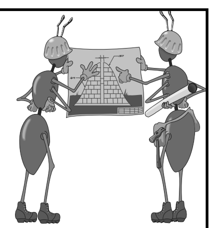
- If defects are written on sticky notes, you will lose track
  - Use Bugzilla (or even just a spreadsheet!)
  - Record any problem that isn’t fixed right away
  - Track to resolution to make sure it is fixed
    - Or marked as “we’re not going to fix this one”
- Ideally, identify root causes to fix them
  - Many times root cause reveals a process problem (e.g., skipped design review, or ineffective testing)
- Start counting defects at a defined place in process
- Do some data analysis to find common problems
  - If a particular module is a <u>Bug Farm</u>, throw it away and start over instead of forever fixing yet another bug

Defect Prioritization

- Prioritize defects based on importance to company
  - Not just how spectacular the results are
  - A risk matrix may be helpful:

BUG FIX PRIORITY		Probability				
		Very High	High	Medium	Low	Very Low
Consequence	Very High	Very High	Very High	Very High	High	High
	High	Very High	High	High	Medium	Medium
	Medium	High	High	Medium	Medium	Low
	Low	High	Medium	Medium	Low	Very Low
	Very Low	Medium	Low	Low	Very Low	Very Low

Run-Time Instrumentation



(#22 No run-time fault instrumentation nor error logs)

- If you get a returned unit that works OK…
  - Was it a software defect you can't reproduce?
  - Was it an intermittent hardware defect?
  - Was it a distributor reducing inventory size?
- Run-time instrumentation gives you a clue
  - Log reboots and up-times
  - Log assertion violations "assert(X==Y);"
  - Log fault codes or other anomalies

Related Defect/Issue Topics

(#23 Defect resolution for 3rd party software)

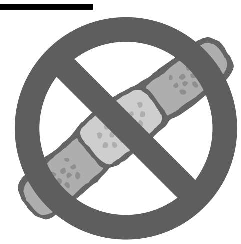
- If a 3rd party package has a bug, what happens?
- What happens to your fixes for new versions?
  - What if it is a new "feature" and not really a bug?

(#24 Disaster recovery not tested)

- If you need to rebuild an old system, can you?
  - Are you sure the files are still there?
  - When was the last time you tested recovery?

Design For Quality Attributes

- Build quality in; don't add it on
  - Performance (better algorithms) and other attributes



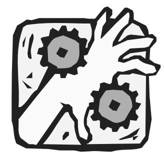
(#25 Insufficient consideration of reliability/availability)

- How often is your software allowed to crash?
  - "Never" is unrealistic
  - Is quick reboot good enough to keep running?
- Use basic techniques to improve reliability
  - Periodic reboot (especially if you allow "malloc")
  - Watchdog timer
  - Improve software quality with good testing & reviews

Safety

(#26 Insufficient consideration of safety)

- A mishap usually involves uncontrolled release of energy
  - Most embedded systems have actuators…
  - … so in principle could result in a mishap
- Thought experiment:
  - Suppose you intentionally tried to cause an accident by writing malicious software
  - Could you bypass hardware safeties with software?
  - If you could, you need to address safety



- Lots of details to get safety right. Short version:
  - Establish a Safety Integrity Level (SIL) based on risks
  - Follow procedures to design to that SIL
  - Examples: IEC 61508 (process), ISO 26262 (automotive)

[IEC 61508-3]	Technique/Measure*	Ref	SIL1	SIL2	SIL3	SIL4
1	Fault detection and diagnosis	C.3.1	---	R	HR	HR
2	Error detecting and correcting codes	C.3.2	R	R	R	HR
3a	Failure assertion programming	C.3.3	R	R	R	HR
3b	Safety bag techniques	C.3.4	---	R	R	R
3c	Diverse programming	C.3.5	R	R	R	HR
3d	Recovery block	C.3.6	R	R	R	R
3e	Backward recovery	C.3.7	R	R	R	R
3f	Forward recovery	C.3.8	R	R	R	R
3g	Re-try fault recovery mechanisms	C.3.9	R	R	R	HR
3h	Memorising executed cases	C.3.10	---	R	R	HR
4	Graceful degradation	C.3.11	R	R	HR	HR
5	Artificial intelligence - fault correction	C.3.12	---	NR	NR	NR
6	Dynamic reconfiguration	C.3.13	---	NR	NR	NR
7a	Structured methods including for example, JSD, MASCOT, SADT and Yourdon.	C.2.1	HR	HR	HR	HR
7b	Semi-formal methods	Table B.7	R	R	HR	HR
7c	Formal methods including for example, CCS, CSP, HOL, LOTOS, OBJ, temporal logic, VDM and Z	C.2.4	---	R	R	HR
8	Computer-aided specification tools	B.2.4	R	R	HR	HR

Security

(#27 Insufficient consideration of security)

(#28 No IP protection plan)

- Most embedded systems have security concerns
  - If there is money to be made or reputation to be gained, attacks will *eventually* happen
  - If someone wants to reverse engineer your product they will
    - (At surprisingly low cost)

Hacker Disables More Than 100 Cars Remotely

By Kevin Poulsen | March 17, 2010 | 1:52 pm | Categories: Breaches, Crime, Cybersecurity, Hacks and Cracks

More than 100 drivers in Austin, Texas found their cars disabled or the horns honking out of control, after an intruder ran amok in a web-based vehicle-immobilization system normally used to get the attention of consumers delinquent in their auto payments.

Police with Austin's High Tech Crime Unit on Wednesday arrested 20-year-old Omar Ramos-Lopez, a former Texas Auto Center employee who was laid off last month, and allegedly sought revenge by bricking the cars sold from the dealership's fo lots.

Hackers Crack Into Texas Road Sign, Warn of Zombies Ahead

Thursday, January 29, 2009
FOX NEWS

By Joshua Rhett Miller

E-Mail | Print | S

Texas Dept. of Transportation officials confirm a portable traffic sign at Lamar Boulevard and West 15th Street in Austin was hacked into last week.

Transportation officials in Tex are scrambling to prevent hac from changing messages on r road signs after one sign in Au was altered to read, "Zombies Ahead."

News

Siemens: Stuxnet worm hit industrial systems

By Robert McMillan
September 14, 2010 01:17 PM ET | Comments (4) | Recommended (21) | Shar

IDG News Service - A sophisticated worm designed to steal industrial secrets and disrupt operations has infected at least 14 plants, according to Siemens.

Called Stuxnet, the worm was discovered in July when researchers at VirusBlokAda found it on computers in Iran. It is one of the most sophisticated and unusual pieces of malicious software ever created -- the worm leveraged a previously unknown Windows vulnerability (now patched) that allowed it to spread from computer to computer, typically via USB sticks.

The worm, designed to attack Siemens industrial control systems, has not spread widely. However, it has affected a number of Siemens plants, according to company spokesman Simon Wieland. "We detected the virus in the SCADA [supervisory control and data acquisition] systems of 14 plants in operation but without any malfunction of process and production and without any damage," he said in an e-mail message.

Polish Teen Hacks His City's Trams, Chaos Ensues

By Chuck Squatriglia | January 11, 2008 | 4:29:44 PM | Categories: Public Transit

A teenager in Lodz, Poland hacked the city's tram system with a homemade transmitter that tripped rail switches and redirected trains, a prank that derailed four trams and injured a dozen people.

According to reports in the Register and the Telegraph, the 14-year-old boy - described by his teachers as an electronics genius (Gee- you think?) - spent months studying the city's rail

Security Plan

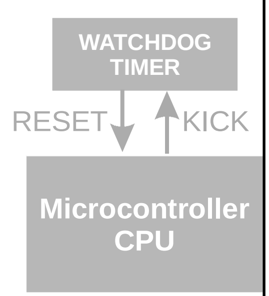
- Written plan for security approach
  - Goals
    - What does being secure mean for you?
  - Plausible attacks & consequences
  - Countermeasures and monitoring
  - Update/patch strategy
- Do-it-yourself security is a bad idea
  - Bake-your-own crypto is an especially bad idea
  - Security via obscurity doesn't work
- Avoid: modems with unlisted numbers, home-made crypto, home-made secret key generators, secret master keys, secret network unlock incantations, head-in-the-sand

Learn today. Design tomorrow.
ESC
Silicon Valley • May 2 - 5, 2011
McEnery Convention Center • San Jose

<u>Use Watchdog Timer Correctly</u>

(#29 No or incorrect use of watchdog timers)

- Common mistakes:
  - Watchdog turned off
  - Watchdog hooked up to HW counter/timer
  - Watchdog kicked by low priority ISR (what about main loop?)
  - Watchdog kicked inside loop of a single task



- Key best practices
  - Kick watchdog in only one place in the code
  - If ***<u>any task</u>*** hangs, don't kick watchdog

Learn today. Design tomorrow.
ESC
Silicon Valley • May 2 - 5, 2011
McEnery Convention Center • San Jose

Incorrect Watchdog Timer Use

CAUTION
INCORRECT CODE

- Consider a preemptive tasking system
  - Assume there is a watchdog timer (a COP timer)
  - kick() restarts the watchdog time at initial value

```
void Task0(void) {..Do stuff..; Kick(); …more… ;}
void Task1(void) {..Do stuff..; Kick(); …more… ;}
void Task2(void) {..Do stuff..; Kick(); …more… ;}
void Task3(void) {..Do stuff..; Kick(); …more… ;}
```

  - Some tasks might be ISRs, others might be RTOS tasks

- What's wrong with the above approach?

Better Multi-Tasking Watchdog Approach

```
void Task0(void) { .. Do stuff..; Alive(0x1); …more… ;}
void Task1(void) { .. Do stuff..; Alive(0x2); …more… ;}
void Task2(void) { .. Do stuff..; Alive(0x4); …more… ;}
void Task3(void) { .. Do stuff..; Alive(0x8); …more… ;}
```

- Main idea – each task sets a bit indicating it has run
  - Separate watchdog monitor task kicks watchdog only when every task reports in
  - Needs to be modified to account for task periods, but this is the basic idea

```
uint16 WatchFlag = 0;
void Alive(uint16 x)
{ SEI();     // Disable Interrupts
  WatchFlag |= x;
  CLI();     // Enable Interrupts
}  // set task's "I'm Alive" bit

void TaskW(void)   // run periodically
{ if (WatchFlag == 0x0F) // if all tasks alive
  { Kick();              // kick watchdog
    WatchFlag = 0;       // erase flags
  }
}
```

System Reset Gotchas

(#30 Inadequate system reset approach)

- Is there a way to reset your system manually?
  - If there is a carry-through capacitor, how long does it last?
- Do all the outputs reset to a safe value?
  - What if the system freezes during initialization?
  - Do you sample all sensors to get new values?
  - Do you re-init all integrators to warm up control loops after a reset?
- What if reset reboots repeatedly (yo-yo mode)?
  - Track reboot frequency (log time while up)
  - After repeated reboots, need a Plan B

http://en.wikipedia.org/wiki/Image:1791-Yo-Yo-Bandalore.jpg

Manage Change

(#31 High requirements churn)

- If requirements change every day, you'll never finish
  - But, requirements change is a fact of life
- Pick a model compatible with your change rate
  - E.g., incremental development for high change rates
- Ensure that cost of change is accounted for
  - Almost no change is truly "free"
  - Extend schedule, increase cost, or delete other features
- Impose a freeze date
  - At some point changes go into next version
- Identify a "Change Control Board" – yes/no decision owner
  - Make sure they are incentivized in a sensible manner
  - Director of Marketing makes a poor CCB

Version Management

(#32 No version control)

- Make sure you can recreate any version
  - Unroll changes
  - Create old version for bug recreation & fixes
  - That includes tools used to build old version

(#33 No backward compatibility plan)

- If you have many products, do they inter-operate?
  - Combinatorial explosion of many old versions
  - Have a policy, e.g., support last 2-3 versions

Learn today. Design tomorrow. ESC Silicon Valley • May 2 - 5, 2011 McEnery Convention Center • San Jose

Software Updates

(#34 No software update plan)

- Your software will have bugs!
  - How do users know they need patches?
- How are patches deployed?
  - Do patches require a service call?
  - How much will it cost to US Mail SD cards with patches to all your customers?
  - Can the user brick the system by botching a patch?
  - Are you worried about malicious fake patches?
  - Do patch connections open security vulnerabilities?

Learn today. Design tomorrow. ESC Silicon Valley • May 2 - 5, 2011 McEnery Convention Center • San Jose

Processes Change Too

(#35 Lessons learned not being recorded)

- You only get smarter if you pay attention
  - Hold an end-of-cycle retrospective
- Tribal wisdom isn't inherited
  - It must be taught
  - Do you set aside time to teach all of it?
- Wisdom only sticks if you write it down
  - If you found something broken, fix the process
  - If you have a new idea, update the process
  - Jettison stuff that isn't working; augment stuff that is
    - For example, design review checklists, coding style, test plans

Don't Think Software Is Free

(#36 Acting as if software is free)

- Good software is expensive
  - Bad software is even more expensive … eventually
- Embedded software is ballpark $20-$40 /SLOC
  - Productivity is usually 1-2 Source Line of Code/hr
- Examples of pretending software is free
  - Add a new function; keep end date the same
  - Lose a team member; keep end date the same
  - Optimize for a smaller CPU; keep identical budget
  - Manage by head count and not project size
  - Set aside zero budget for old-version maintenance
  - Ignoring effort to port code & interact with "free" software community to obtain maintenance

"Free" Tools Aren't Free

- "I'll spend a month porting a free compiler"
  - Is that really worth ~$10K of cost savings?
  - Even if the "free" compiler is really good?
- "I'll write my own RTOS and save money"
  - 5000 SLOC @ $40/line = $200,000
  - You're dreaming if you think RTOS code is only $40/SLOC if you really want it to work
  - And, most of us aren't good enough to get it right

(#37 Use of cheap tools instead of good ones)

- We can't afford a good compiler, so we use a cheap one
  - … with terrible compiler warnings
  - … with bugs to work around
  - … that is hard to debug with … etc.

Developer Burnout

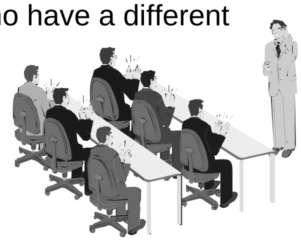
(#38 High turnover and developer overload)

- If you abuse your developers:
  - By assuming they can write 2x the code at 1x the cost
  - By jerking them around with requirement churn
  - By not giving them the time to improve skills & process
  - ….
- Don't be surprised if they bail out
  - And you have no idea what is in the code
  - And you have lost your tribal knowledge
  - …

Even Smart People Need Training

(#39 No training for managing outsource relationships)

- If you are off-shoring effort, need training for
  - Better process to create clean hand-offs
  - Management of outsource partners who have a different business model than you do
  - Cultural differences

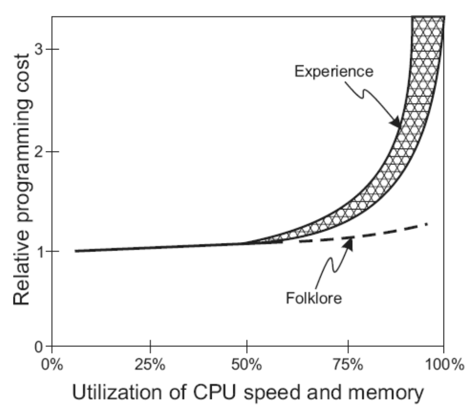


- Also need training for:
  - Design reviews and other helpful non-offshore processes
  - Deeper embedded systems skills, especially for domain experts who are self-taught at computers

Have Slack Resources

(#40 Resources too full)

- For typical embedded hardware/software costs:
- If production run is less than 1 MILLION units
  - Resources should be no more than 80% full
- If production run is less than 10K units
  - Resources should be no more than 50% full



(Source: Barry Boehm, 1975)

Zero Is The Right Amount of Assembly Code

(#41 Too much assembly language)

- It takes 4-5 lines of assembler to match 1 C line
  - Cost scales proportional to source code size
    - Cost/line relatively independent of language
  - Bug rate scales at least proportional to code size
    - Probably higher for assembly – no variable typing
  - Portability is severely reduce in assembly
- Assembly costs 4x-5x as much as C

- Unless software is free, get a bigger CPU
  - (Don't forget: #36 Acting as if software is free)

Make Sure You Follow Your Process

(#42 Project schedule not taken seriously)

- Lip service worse than a waste of time
  - Because it fools you into thinking you are making progress

- Which of these scenarios is a problem?
  - Management determined schedule before defining project content
  - # developers determined by head count restrictions rather than size and schedule estimates
  - Developers are running behind … steal time from test
  - Software developers get weekends off to be with their families

Is Your Process Working?

(#43 No Software Quality Assurance (SQA) function)

- QA – Quality Assurance
  - Usually this refers to software testing
  - But, it is only a partial predictor of software quality!
    - Understanding true quality requires understanding process too
- SQA – Software Quality Assurance
  - This is about whether you are following your *process*
  - Did you actually do what you said you'd do?
    - Regardless of how heavy/light that may be
- SQA should be perhaps 6% of your effort
  - Half to define, maintain, train on processes
  - Half to audit, collect metrics, and monitor

About The Dark Side Of SQA

- Avoid SQA "process police" mentality
  - Especially if developers don't see value in the processes
  - But, you still need to see what's really happening
- A "Coach" style can be positive:
  - Help developers define what they actually want to do
  - Help find ways to improve development outcomes
  - Help developers find times when they aren't actually doing what they said they wanted to do
  - Spot quality problems early, before the train wreck
    - Requires taking and monitoring lightweight metrics
  - Give developers cover during time crunches
    - SQA should not sign off if shortcuts were taken on development

An Initial Agenda For Better Quality

- ☑ Hire good people. Process doesn't fix incompetence.
- ☑ Define your process (steps & artifacts) on one page
  - → You can't get there without a map
- ☑ Do peer reviews early, often, and effectively
  - → Biggest bang-for-buck there is
- ☑ Do balanced, planned testing
  - → Define & track coverage
  - → Start test planning & testing before the end
- ☑ Track if your process is healthy
  - → Are you generating all the artifacts in your process?
  - → Is peer review finding about half the bugs?
  - → Are you spending 50%-65% of total project effort on reviews, test, quality, SQA?
  - → Are defects clustering into bug farms (product or process)?

Questions?

- For after-session questions, mail to:
  - Koopman@cmu.edu
  - Please indicate if:
    - It is OK to quote your question on my blog
    - It is OK to mention your full name, just your first name, or call you "anonymous"
- Questions of general interest that I can post onto my blog will receive highest response priority
  - http://BetterEmbSW.BlogSpot.com/