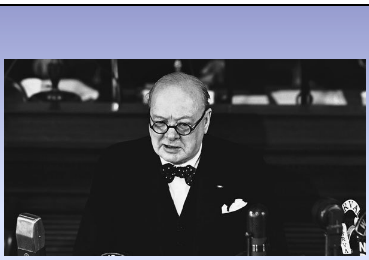
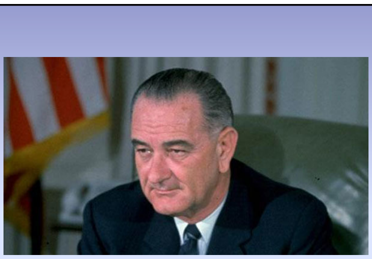


It has been said that democracy is the worst form of government except all those other forms that have been tried from time to time.

Winston Churchill, on the floor of Parliament, November 1947



A man without a vote is a man without protection.

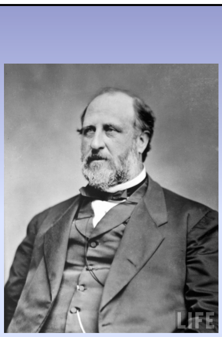
Commonly attributed to President Lyndon Johnson

Voting

- Accurate voting systems are critical in every nation that holds open elections
- Let’s explore a little history

As long as I get to count the votes, what are you going to do about it?

William Magear (“Boss”) Tweed



Boss Tweed

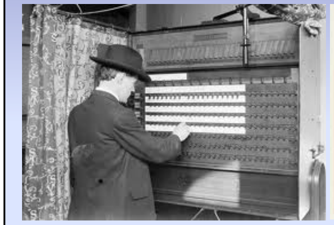
- Tweed was elected to the
  - U.S. House of Representatives in 1852
  - New York County Board of Supervisors in 1858
  - New York State Senate in 1867
- His political influence came from
  - being an appointed member of many boards and commissions
  - his control over political patronage in New York City through Tammany Hall (the Democratic Party political machine)
  - his ability to ensure the loyalty of voters through jobs he could create and control on city-related projects
- He was convicted in 1877 of stealing ~$200 million from taxpayers
- He died in jail in 1878

It's hard not to admire the skill behind Tweed's system... The Tweed ring at its height was an engineering marvel, strong and solid, strategically deployed to control key power points: the courts, the legislature, the treasury and the ballot box. Its frauds had a grandeur of scale and an elegance of structure: money-laundering, profit sharing and organization.

Kenneth D. Ackerman, Tweed biographer

Voting machines: some history

- The U.S. has more contests per ballot than any other country
- Challenge of hand-counting ballots motivated development of machinery to count votes
- First patent for machine useful in general election in U.S. issued in 1881
  - Array of buttons, one row per office, one column per party
  - Interlocks prevented voting for more than one candidate per race
  - Door interlock reset the machine as each voter left the booth
- Challenges with that technology:
  - Poll workers (volunteers) could not verify correct operation of machine
  - Trickery could affect outcome: gears shaved, levers bent slightly, etc.



CAN YOU "GET ALONG"
WITHOUT
AUTOMATIC VOTING MACHINES!
Well, possibly you can . . . just as you "got along" 100 years ago without the automobiles, typewriters, adding machines, and telephones that enable you to operate so efficiently today . . .
BUT!
This Is Today . . . Not 100 Years Ago!
products for the public good
AUTOMATIC
VOTING MACHINE
division of Rockwell Manufacturing Company
Jamestown, N. Y.

Background, cont.

- 2000 election (Bush v. Gore) pointed out problems with election systems
  - Outcome came down to Florida – no clear winner on election night
  - Race was so close that state law required a recount
  - Many legal battles followed: how to count non-clear cut cases
  - Votes were cast on punch cards; some had *hanging chads*, others had an indentation but no hole
  - After one month, U.S. Supreme Court stopped the recount
  - Bush was declared the winner by 537 votes (a margin of 0.009%)
  - Conclusion of many: should move from punch-cards to all-digital systems
  - Subsequent years saw increased use of DREs (direct-recording electronic voting machines)
  - DREs introduced plenty of new problems...

Problems with DREs: 1

- 2000, New Jersey:
  - A DRE was taken out of service after a total of 65 votes were cast
  - After election, it was determined that *none* of the 65 votes was recorded for either the Republican or the Democratic candidate for one office, but 27 votes each were recorded for their running mates
  - Company representative said no votes were lost: all of those 65 voters simply failed to vote for the top two candidates
  - No way to know for sure

Problems with DREs: 2

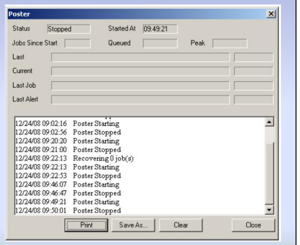
- 2002, Florida:
  - Runoff election decided by 5 votes
  - 78 ballots had no vote recorded
  - Election supervisor claimed that those 78 people simply didn't cast a vote – despite the fact that it was the only contest on the ballot!
  - There was simply no way to know for sure

Problems with DREs: 3

- 2006, Florida:
  - The winning margin in the 13th congressional district was just 369 votes
  - But more than 18,000 ballots from Sarasota county had no vote in the race
  - There was simply no way to know the intents and actions of those voters

Problems with DREs: 4

- 2008, California:
  - Voting system lost about 200 completed ballots in Humboldt County
  - Investigation by CA Secretary of State showed that the GUI for Diebold's vote tabulation system had a button allowing operator to delete audit logs
  - Button was located next to "print" and "save as" buttons
  - But audit logs are required by federal voting system guidelines!
  - System developer warned Diebold in 2001 email against adding clear button, but company ignored those concerns

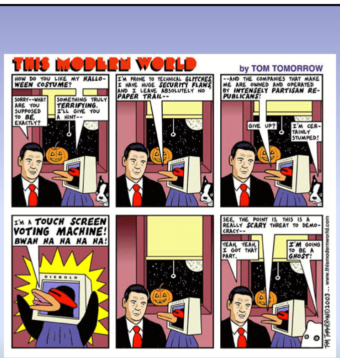


Status: Stopped
Started At: 09:49:21
Jobs Since Start
Queued
Peak

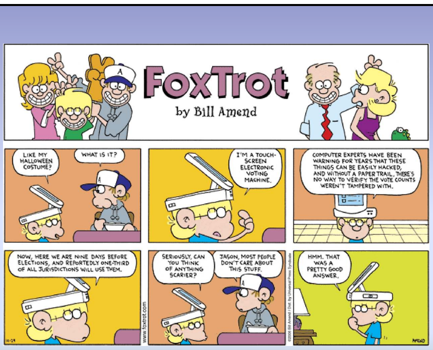
```
12/24/08 09:02:16   Poster Starting
12/24/08 09:02:56   Poster Stopped
12/24/08 09:20:20   Poster Starting
12/24/08 09:21:00   Poster Stopped
12/24/08 09:22:13   Recovering 0 job(s)
12/24/08 09:22:13   Poster Starting
12/24/08 09:22:53   Poster Stopped
12/24/08 09:46:07   Poster Starting
12/24/08 09:46:47   Poster Stopped
12/24/08 09:49:21   Poster Starting
12/24/08 09:50:01   Poster Stopped
```

Problems with DREs: 4 (cont.)

- 2008, California:
  - Problem #2 discovered in investigation: older version of software dropped all votes in the first *deck* of ballots run through the system
  - Company knew about problem #2, but simply instructed officials to begin elections by creating and deleting an empty "deck zero"


THIS MODERN WORLD
by TOM TOMORROW
HOW DO YOU LIKE MY HALLOWEEN COSTUME?
SORRY--WHAT ARE YOU SUPPOSED TO BE, EXACTLY?
SOMETHING TRULY TERRIFYING. I'LL GIVE YOU A HINT--
I'M PRONE TO TECHNICAL GLITCHES, I HAVE HUGE SECURITY FLAWS, AND I LEAVE ABSOLUTELY NO PAPER TRAIL--
--AND THE COMPANIES THAT MAKE ME ARE OWNED AND OPERATED BY INTENSELY PARTISAN REPUBLICANS!
GIVE UP?
I'M CERTAINLY STUMPED!
I'M A TOUCH SCREEN VOTING MACHINE! BWAH HA HA HA HA!
DIEBOLD
SEE, THE POINT IS, THIS IS A REALLY SCARY THREAT TO DEMOCRACY--
YEAH, YEAH, I GOT THAT PART.
I'M GOING TO BE A GHOST!
TOM TOMORROW©2003 ... www.thismodernworld.com

Halloween 2003


FoxTrot
by Bill Amend
LIKE MY HALLOWEEN COSTUME?
WHAT IS IT?
I'M A TOUCH-SCREEN ELECTRONIC VOTING MACHINE.
COMPUTER EXPERTS HAVE BEEN WARNING FOR YEARS THAT THESE THINGS CAN BE EASILY HACKED, AND WITHOUT A PAPER TRAIL, THERE'S NO WAY TO VERIFY THE VOTE COUNTS WEREN'T TAMPERED WITH.
NOW, HERE WE ARE NINE DAYS BEFORE ELECTIONS, AND REPORTEDLY ONE-THIRD OF ALL JURISDICTIONS WILL USE THEM.
SERIOUSLY, CAN YOU THINK OF ANYTHING SCARIER?
JASON, MOST PEOPLE DON'T CARE ABOUT THIS STUFF.
HMM. THAT WAS A PRETTY GOOD ANSWER.
10-29
www.foxtrot.com
©2006 Bill Amend / Dist. by Universal Press Syndicate
AMEND

October 29, 2006

Nisley article: April 2001

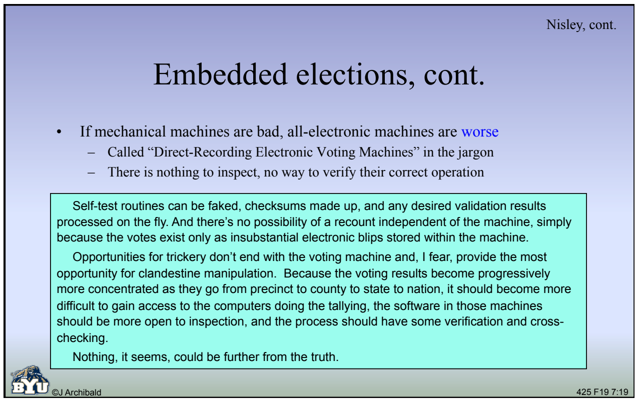
- Simple to build a voting machine that records punched holes, right?
  - What if they aren't punched exactly right?
  - Is such a vote invalid, or can someone determine what voter intended?
- There is much more to voting than tabulating punched cards; there is an entire system to consider, including *political* implications
  - Results combined with others in hierarchy reaching state, national levels
  - Accuracy depends on every step along the way
  - Any errors in transmitting or recording can fatally corrupt the results
- Checks and balances are vital if people are to have trust in the system
  - Current practice: members of major parties allowed to examine all voting materials, inspect voting machines, certify that votes are accurately counted and reported
  - Intent: reduce fraud, while preserving privacy and anonymity of each vote

Source: "Embedded elections", Ed Nisley, Dr. Dobb's Journal, April 2001
(Written shortly after the hanging chad fiasco of the 2000 election.)

Nisley, cont.

Embedded elections

- As process changes with technology, can observation still take place?
  - Can we maintain the critical checks and balances?
- Consider mechanical voting machines:
  - Levers on front, mechanical counters on back, interconnected by rods
  - Powered by same handle that closes privacy curtain
  - Inspectors verify that dials read 0 at start, then simply record totals at end
- How might one affect final tally on such a machine?
  - Gears can be shaved to skip counts, levers slightly bent, etc.
  - Election inspectors are volunteers, not engineers or forensic experts
  - The machines are not torn down for a complete check before election
  - Inspectors can only verify that machine *seems* to be working

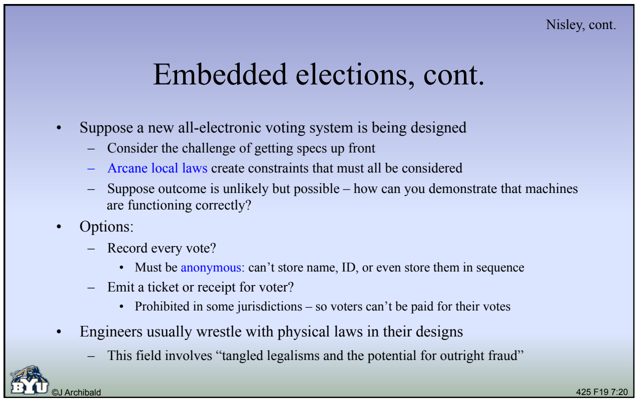


Nisley, cont.

Embedded elections, cont.

- If mechanical machines are bad, all-electronic machines are worse
  - Called "Direct-Recording Electronic Voting Machines" in the jargon
  - There is nothing to inspect, no way to verify their correct operation

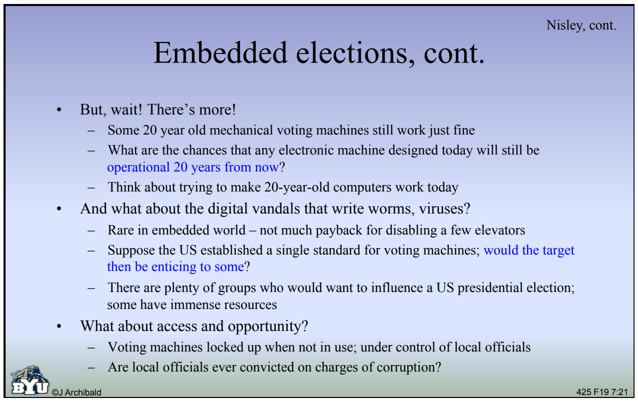
> Self-test routines can be faked, checksums made up, and any desired validation results processed on the fly. And there's no possibility of a recount independent of the machine, simply because the votes exist only as insubstantial electronic blips stored within the machine.
>
> Opportunities for trickery don't end with the voting machine and, I fear, provide the most opportunity for clandestine manipulation. Because the voting results become progressively more concentrated as they go from precinct to county to state to nation, it should become more difficult to gain access to the computers doing the tallying, the software in those machines should be more open to inspection, and the process should have some verification and cross-checking.
>
> Nothing, it seems, could be further from the truth.



Nisley, cont.

Embedded elections, cont.

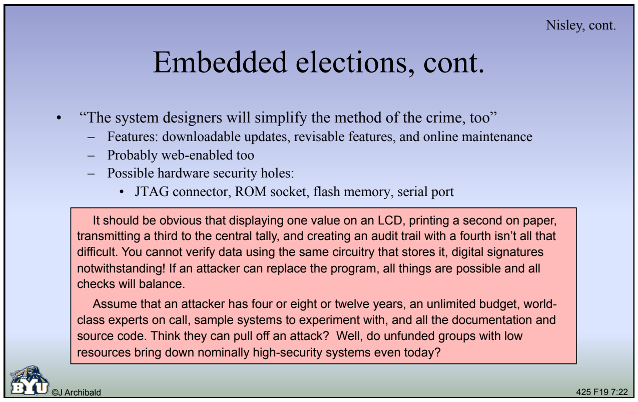
- Suppose a new all-electronic voting system is being designed
  - Consider the challenge of getting specs up front
  - Arcane local laws create constraints that must all be considered
  - Suppose outcome is unlikely but possible – how can you demonstrate that machines are functioning correctly?
- Options:
  - Record every vote?
    - Must be anonymous: can't store name, ID, or even store them in sequence
  - Emit a ticket or receipt for voter?
    - Prohibited in some jurisdictions – so voters can't be paid for their votes
- Engineers usually wrestle with physical laws in their designs
  - This field involves "tangled legalisms and the potential for outright fraud"



Nisley, cont.

Embedded elections, cont.

- But, wait! There's more!
  - Some 20 year old mechanical voting machines still work just fine
  - What are the chances that any electronic machine designed today will still be operational 20 years from now?
  - Think about trying to make 20-year-old computers work today
- And what about the digital vandals that write worms, viruses?
  - Rare in embedded world – not much payback for disabling a few elevators
  - Suppose the US established a single standard for voting machines; would the target then be enticing to some?
  - There are plenty of groups who would want to influence a US presidential election; some have immense resources
- What about access and opportunity?
  - Voting machines locked up when not in use; under control of local officials
  - Are local officials ever convicted on charges of corruption?

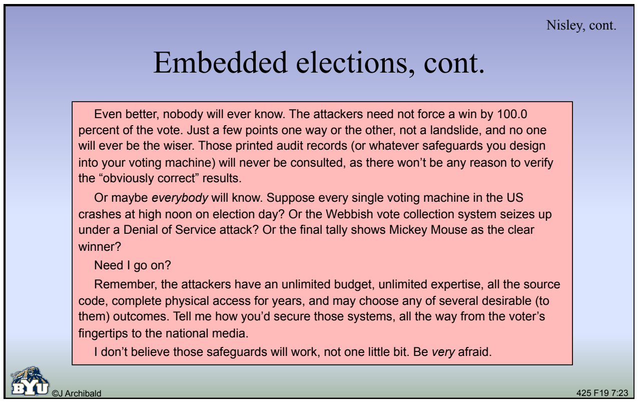


Nisley, cont.

Embedded elections, cont.

- "The system designers will simplify the method of the crime, too"
  - Features: downloadable updates, revisable features, and online maintenance
  - Probably web-enabled too
  - Possible hardware security holes:
    - JTAG connector, ROM socket, flash memory, serial port

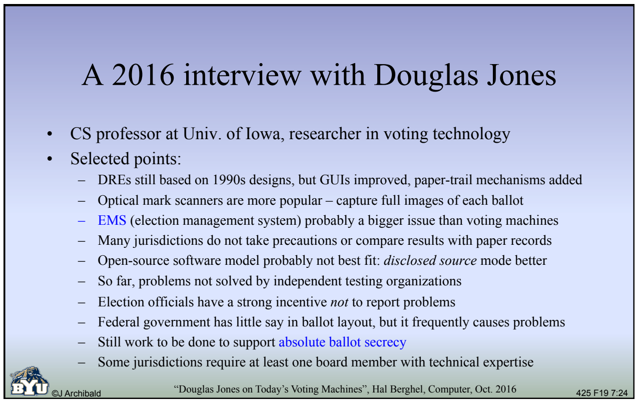
> It should be obvious that displaying one value on an LCD, printing a second on paper, transmitting a third to the central tally, and creating an audit trail with a fourth isn't all that difficult. You cannot verify data using the same circuitry that stores it, digital signatures notwithstanding! If an attacker can replace the program, all things are possible and all checks will balance.
>
> Assume that an attacker has four or eight or twelve years, an unlimited budget, world-class experts on call, sample systems to experiment with, and all the documentation and source code. Think they can pull off an attack? Well, do unfunded groups with low resources bring down nominally high-security systems even today?



Nisley, cont.

Embedded elections, cont.

> Even better, nobody will ever know. The attackers need not force a win by 100.0 percent of the vote. Just a few points one way or the other, not a landslide, and no one will ever be the wiser. Those printed audit records (or whatever safeguards you design into your voting machine) will never be consulted, as there won't be any reason to verify the "obviously correct" results.
>
> Or maybe *everybody* will know. Suppose every single voting machine in the US crashes at high noon on election day? Or the Webbish vote collection system seizes up under a Denial of Service attack? Or the final tally shows Mickey Mouse as the clear winner?
>
> Need I go on?
>
> Remember, the attackers have an unlimited budget, unlimited expertise, all the source code, complete physical access for years, and may choose any of several desirable (to them) outcomes. Tell me how you'd secure those systems, all the way from the voter's fingertips to the national media.
>
> I don't believe those safeguards will work, not one little bit. Be *very* afraid.



A 2016 interview with Douglas Jones

- CS professor at Univ. of Iowa, researcher in voting technology
- Selected points:
  - DREs still based on 1990s designs, but GUIs improved, paper-trail mechanisms added
  - Optical mark scanners are more popular – capture full images of each ballot
  - EMS (election management system) probably a bigger issue than voting machines
  - Many jurisdictions do not take precautions or compare results with paper records
  - Open-source software model probably not best fit: *disclosed source* mode better
  - So far, problems not solved by independent testing organizations
  - Election officials have a strong incentive *not* to report problems
  - Federal government has little say in ballot layout, but it frequently causes problems
  - Still work to be done to support absolute ballot secrecy
  - Some jurisdictions require at least one board member with technical expertise

"Douglas Jones on Today's Voting Machines", Hal Berghel, Computer, Oct. 2016

Jones

- Will internet voting solve our problems?

> Coming up with “best practices for internet voting” is like coming up with “best practices for drunk driving.” You really don’t want to go there.

Epstein: A voting machine’s demise

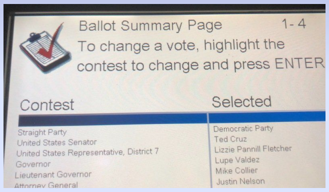
- Case study of voting machine that was still in use in 2014
- Selected points:
  - Based on Windows XP Embedded, used Wi-Fi to configure machines, summarize results
  - No OS patches installed since 2004
  - Used WEP wireless encryption scheme, despite being declared obsolete in 2004
  - WEP key hardwired to “abcde”
  - Windows administrator password set to “admin” – no interface provided to change it
  - Database encrypted using weak scheme with hardwired key
  - No logs or cryptographic checksums

> This collection of security flaws is so severe that if an election was held using the AVS WinVote, and it wasn’t hacked, it was only because no one tried.

“Weakness in Depth: A Voting Machine’s Demise”, Jeremy Epstein, IEEE Security & Privacy, May/June 2015

Were problems fixed in 2018?

- Associate Press:
  - Political activist Leah McElrath voted early in Texas on Oct. 22, 2018
  - She reviewed her electronic ballot, saw this on her screen and took a photo
  - Despite her straight party vote (for Democrats), the machine reported a vote for Republican Ted Cruz

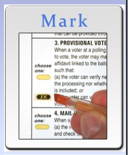


Ballot Summary Page 1 - 4
To change a vote, highlight the contest to change and press ENTER

Contest	Selected
Straight Party	Democratic Party
United States Senator	Ted Cruz
United States Representative, District 7	Lizzie Pannill Fletcher
Governor	Lupe Valdez
Lieutenant Governor	Mike Collier
Attorney General	Justin Nelson

Can better technology help?

- Consider **Scantegrity**, an open source, optical scan system developed by researchers from MIT and elsewhere; impetus was NSF-sponsored competition
- Used since 2009
  - Individual voters can verify that their ballots got counted
  - Anyone can independently audit the results
- Voters make selections on paper ballot using special pens
  - Ink in pen reacts with ink in oval, turning it black and revealing unique three-letter code
  - Voters record unique ballot serial number and three-letter code
  - Codes are generated cryptographically and are different on every ballot
- After polls close, voters can go to the election office website, type in serial number and see a rendition of a ballot, showing the codes for their votes
  - Voters can verify that their ballots were included in the final tally
  - Designers claim problems can be identified if just 3 to 5 percent of voters verify their votes

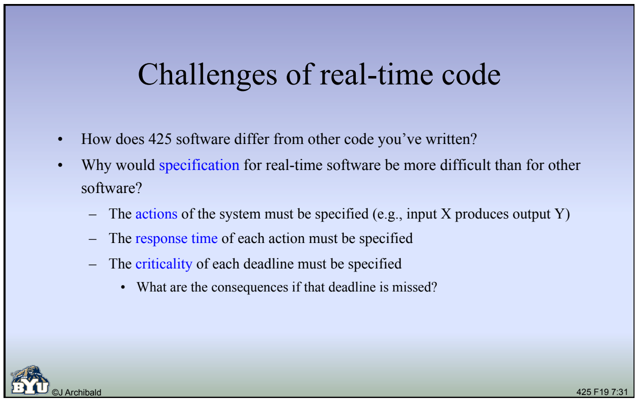


Discussion

- What are consequences if citizens do not believe in election outcome?
- Is there evidence that other actors want to undermine confidence in US elections?
- Can technology ever result in perfectly secure systems?
- Are problems solved if we
  - vote by mail?
  - vote using the Internet?
  - build a system on top of the Internet of Things?

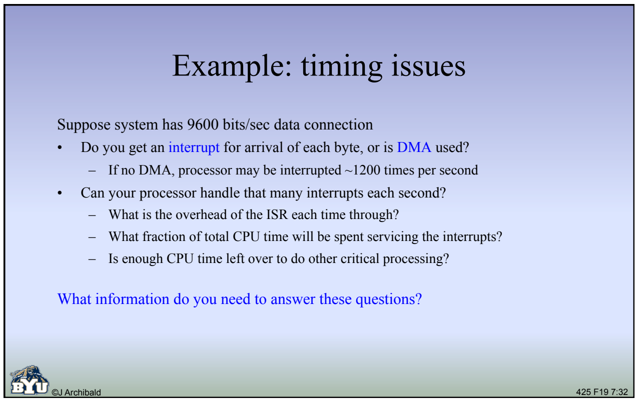
Chapter 8: Design principles

- You’ve experienced some of the challenges of creating concurrent software
- Chapter 8 focuses on design principles for RTOS application code
- You would be wise to apply these principles in Lab 8
- Principles are not hard and fast rules; sometimes you have to break them to get your system to work
- What then is purpose of “rules”?
  - They serve as a useful guide
  - Violations draw attention to potentially dangerous aspects of a design
  - They may help designers discover safer alternatives



Challenges of real-time code

- How does 425 software differ from other code you've written?
- Why would specification for real-time software be more difficult than for other software?
  - The actions of the system must be specified (e.g., input X produces output Y)
  - The response time of each action must be specified
  - The criticality of each deadline must be specified
    - What are the consequences if that deadline is missed?

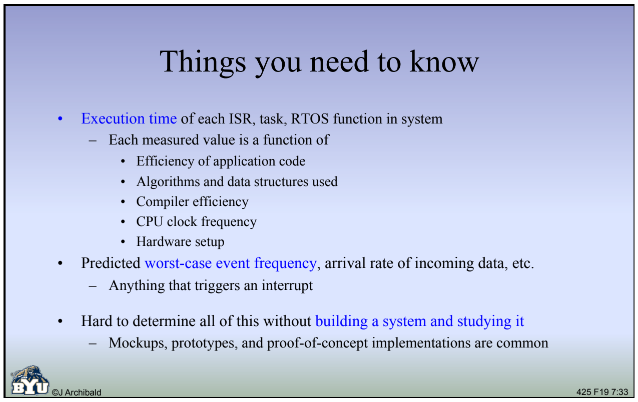


Example: timing issues

Suppose system has 9600 bits/sec data connection

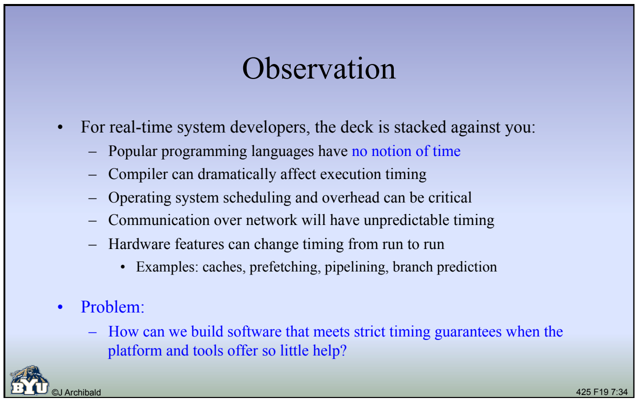
- Do you get an interrupt for arrival of each byte, or is DMA used?
  - If no DMA, processor may be interrupted ~1200 times per second
- Can your processor handle that many interrupts each second?
  - What is the overhead of the ISR each time through?
  - What fraction of total CPU time will be spent servicing the interrupts?
  - Is enough CPU time left over to do other critical processing?

What information do you need to answer these questions?



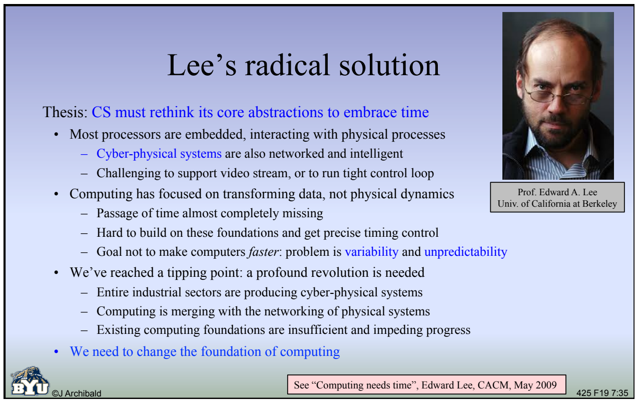
Things you need to know

- Execution time of each ISR, task, RTOS function in system
  - Each measured value is a function of
    - Efficiency of application code
    - Algorithms and data structures used
    - Compiler efficiency
    - CPU clock frequency
    - Hardware setup
- Predicted worst-case event frequency, arrival rate of incoming data, etc.
  - Anything that triggers an interrupt
- Hard to determine all of this without building a system and studying it
  - Mockups, prototypes, and proof-of-concept implementations are common



Observation

- For real-time system developers, the deck is stacked against you:
  - Popular programming languages have no notion of time
  - Compiler can dramatically affect execution timing
  - Operating system scheduling and overhead can be critical
  - Communication over network will have unpredictable timing
  - Hardware features can change timing from run to run
    - Examples: caches, prefetching, pipelining, branch prediction
- Problem:
  - How can we build software that meets strict timing guarantees when the platform and tools offer so little help?



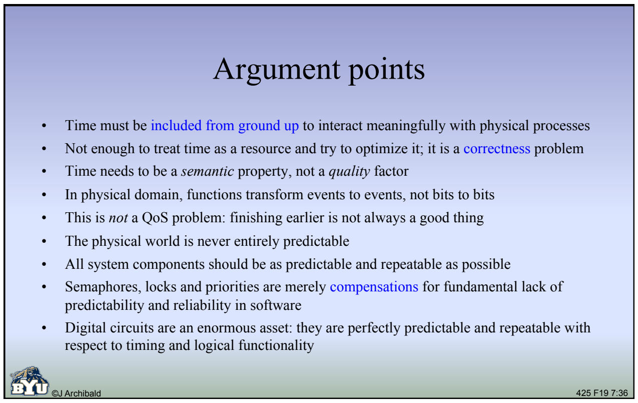
Lee's radical solution

Prof. Edward A. Lee
Univ. of California at Berkeley

Thesis: CS must rethink its core abstractions to embrace time

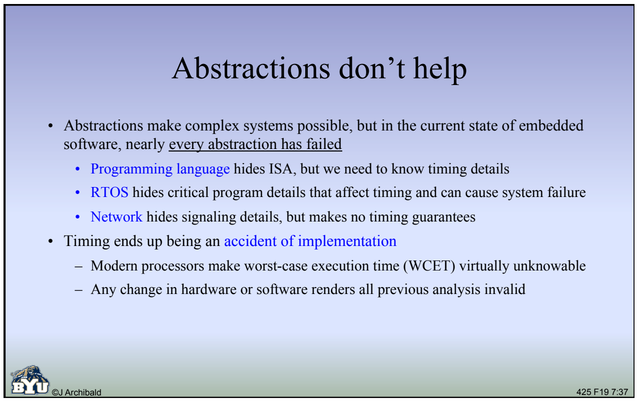
- Most processors are embedded, interacting with physical processes
  - Cyber-physical systems are also networked and intelligent
  - Challenging to support video stream, or to run tight control loop
- Computing has focused on transforming data, not physical dynamics
  - Passage of time almost completely missing
  - Hard to build on these foundations and get precise timing control
  - Goal not to make computers *faster*: problem is variability and unpredictability
- We've reached a tipping point: a profound revolution is needed
  - Entire industrial sectors are producing cyber-physical systems
  - Computing is merging with the networking of physical systems
  - Existing computing foundations are insufficient and impeding progress
- We need to change the foundation of computing

See "Computing needs time", Edward Lee, CACM, May 2009



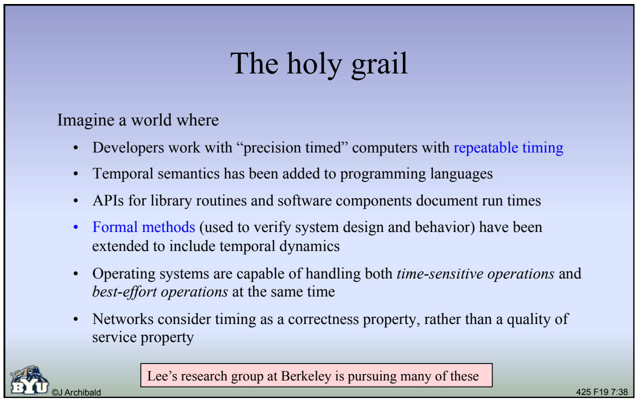
Argument points

- Time must be included from ground up to interact meaningfully with physical processes
- Not enough to treat time as a resource and try to optimize it; it is a correctness problem
- Time needs to be a *semantic* property, not a *quality* factor
- In physical domain, functions transform events to events, not bits to bits
- This is *not* a QoS problem: finishing earlier is not always a good thing
- The physical world is never entirely predictable
- All system components should be as predictable and repeatable as possible
- Semaphores, locks and priorities are merely compensations for fundamental lack of predictability and reliability in software
- Digital circuits are an enormous asset: they are perfectly predictable and repeatable with respect to timing and logical functionality



Abstractions don’t help

- Abstractions make complex systems possible, but in the current state of embedded software, nearly every abstraction has failed
  - Programming language hides ISA, but we need to know timing details
  - RTOS hides critical program details that affect timing and can cause system failure
  - Network hides signaling details, but makes no timing guarantees
- Timing ends up being an accident of implementation
  - Modern processors make worst-case execution time (WCET) virtually unknowable
  - Any change in hardware or software renders all previous analysis invalid

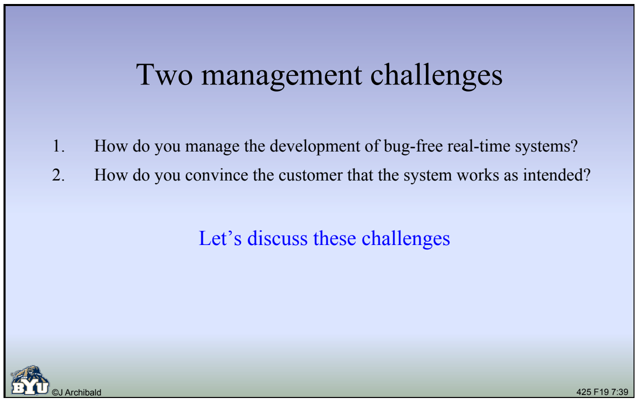


The holy grail

Imagine a world where

- Developers work with “precision timed” computers with repeatable timing
- Temporal semantics has been added to programming languages
- APIs for library routines and software components document run times
- Formal methods (used to verify system design and behavior) have been extended to include temporal dynamics
- Operating systems are capable of handling both *time-sensitive operations* and *best-effort operations* at the same time
- Networks consider timing as a correctness property, rather than a quality of service property

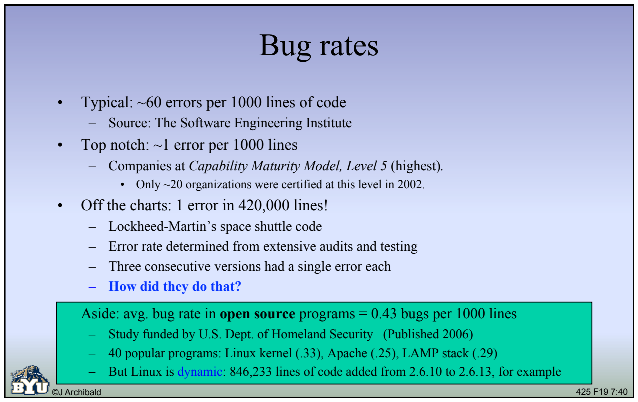
Lee’s research group at Berkeley is pursuing many of these



Two management challenges

1. How do you manage the development of bug-free real-time systems?
2. How do you convince the customer that the system works as intended?

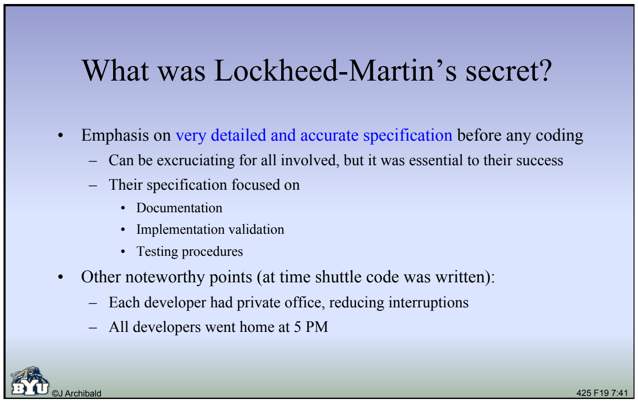
Let’s discuss these challenges



Bug rates

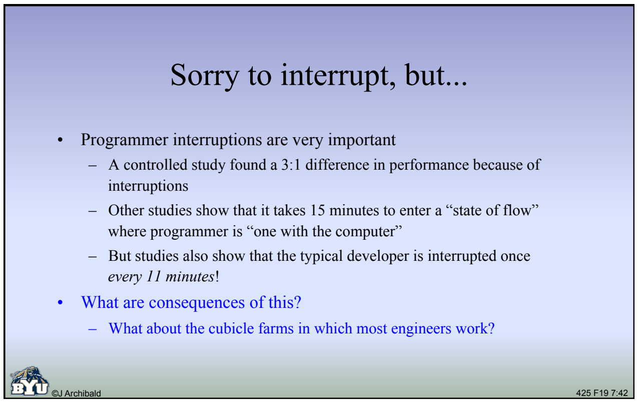
- Typical: ~60 errors per 1000 lines of code
  - Source: The Software Engineering Institute
- Top notch: ~1 error per 1000 lines
  - Companies at *Capability Maturity Model, Level 5* (highest).
    - Only ~20 organizations were certified at this level in 2002.
- Off the charts: 1 error in 420,000 lines!
  - Lockheed-Martin’s space shuttle code
  - Error rate determined from extensive audits and testing
  - Three consecutive versions had a single error each
  - **How did they do that?**

Aside: avg. bug rate in **open source** programs = 0.43 bugs per 1000 lines
- Study funded by U.S. Dept. of Homeland Security (Published 2006)
- 40 popular programs: Linux kernel (.33), Apache (.25), LAMP stack (.29)
- But Linux is dynamic: 846,233 lines of code added from 2.6.10 to 2.6.13, for example



What was Lockheed-Martin’s secret?

- Emphasis on very detailed and accurate specification before any coding
  - Can be excruciating for all involved, but it was essential to their success
  - Their specification focused on
    - Documentation
    - Implementation validation
    - Testing procedures
- Other noteworthy points (at time shuttle code was written):
  - Each developer had private office, reducing interruptions
  - All developers went home at 5 PM



Sorry to interrupt, but...

- Programmer interruptions are very important
  - A controlled study found a 3:1 difference in performance because of interruptions
  - Other studies show that it takes 15 minutes to enter a “state of flow” where programmer is “one with the computer”
  - But studies also show that the typical developer is interrupted once *every 11 minutes*!
- What are consequences of this?
  - What about the cubicle farms in which most engineers work?

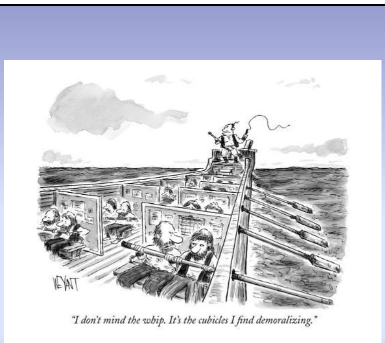
Productivity

- Study from *Peopleware*, DeMarco and Lister, Dorset House Publishing, 1987
- Authors conducted extensive coding competitions for teams: how well they solved standard set of software problems
- Results:
  - Average of top quartile outperformed average of bottom quartile by ~3x!
  - Performance highly correlated with environment; little correlation with experience

	1st Quartile	4th Quartile
Dedicated workspace	78 sq ft	46 sq ft
Is it quiet?	57% yes	29% yes
Is it private?	62% yes	19% yes
Can you turn off phone?	52% yes	10% yes
Can you divert calls?	76% yes	19% yes
Frequent interruptions?	38% yes	76% yes

Fun facts about cubicles

- Dilbert cartoons refer to cubicles as "anti-productivity pods"
- 40M North Americans work in cubicles in 2019
- In 1994, workers had 90 ft^2 on average. This fell to 75 ft^2 by 2010.
- Cubicles block sunlight and result in poor ventilation; studies tie both to decreased productivity and to an increase in sick leave
- People in cubicles with higher partitions work more slowly
- Robert Propst was credited with invented the cubicle, but before his death (2000) he railed against them, calling them "monolithic insanity"
- Intel recently decided to rethink its offices when it was determined that 60% of cubicles were empty most of the time
- The trend is to open-plan areas, with fewer desks than people



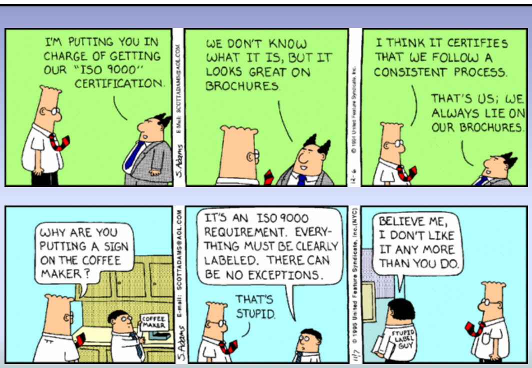
"I don't mind the whip. It's the cubicles I find demoralizing."

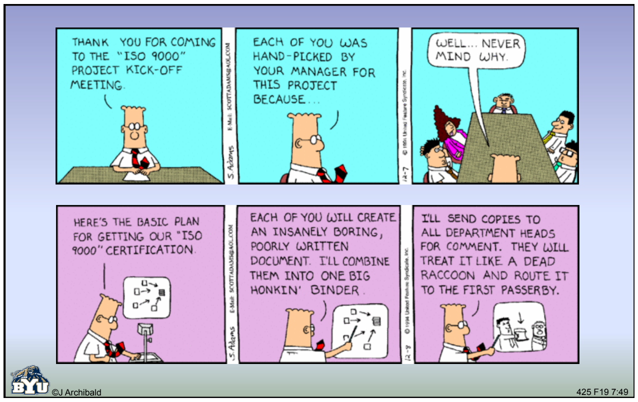
Surviving cubicles

- Recommendations to minimize negative impact
  - Wear headphones, use music to drown out noise
  - Turn the phone off
  - Know and use your most productive hours
  - Disable the email
  - Put a curtain across the opening (some even buy a door!)
  - One website shows how to make a fake window
  - Customize the space: wallpaper, rug, chandelier...

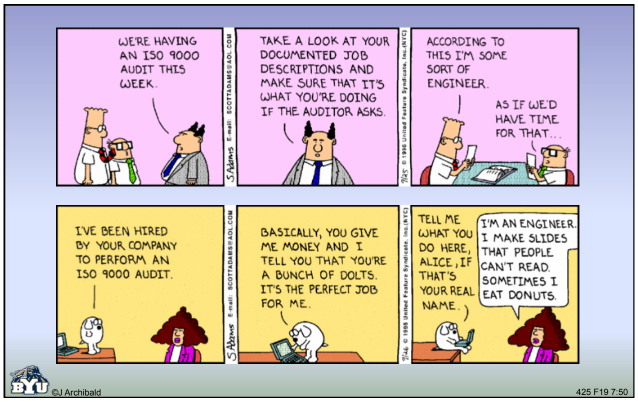
Motivation challenge

- How does typical engineer or programmer feel about spending a lot of time doing the tasks below?
  - Specification
  - Documentation
  - Development
  - Testing
- Let's turn to perhaps the highest technical authority in the land for some insight...

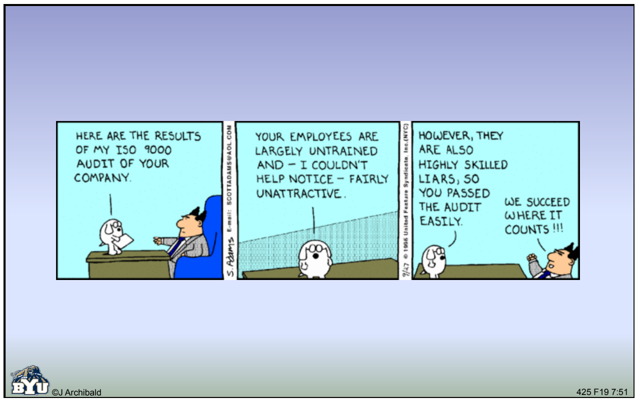

I'M PUTTING YOU IN CHARGE OF GETTING OUR "ISO 9000" CERTIFICATION.
WE DON'T KNOW WHAT IT IS, BUT IT LOOKS GREAT ON BROCHURES.
I THINK IT CERTIFIES THAT WE FOLLOW A CONSISTENT PROCESS.
THAT'S US; WE ALWAYS LIE ON OUR BROCHURES.
WHY ARE YOU PUTTING A SIGN ON THE COFFEE MAKER?
COFFEE MAKER
IT'S AN ISO 9000 REQUIREMENT. EVERY-THING MUST BE CLEARLY LABELED. THERE CAN BE NO EXCEPTIONS.
THAT'S STUPID.
BELIEVE ME, I DON'T LIKE IT ANY MORE THAN YOU DO.
STUPID LABEL GUY
S.Adams


THANK YOU FOR COMING TO THE "ISO 9000" PROJECT KICK-OFF MEETING.
EACH OF YOU WAS HAND-PICKED BY YOUR MANAGER FOR THIS PROJECT BECAUSE...
WELL... NEVER MIND WHY.
HERE'S THE BASIC PLAN FOR GETTING OUR "ISO 9000" CERTIFICATION.
EACH OF YOU WILL CREATE AN INSANELY BORING, POORLY WRITTEN DOCUMENT. I'LL COMBINE THEM INTO ONE BIG HONKIN' BINDER.
I'LL SEND COPIES TO ALL DEPARTMENT HEADS FOR COMMENT. THEY WILL TREAT IT LIKE A DEAD RACCOON AND ROUTE IT TO THE FIRST PASSERBY.

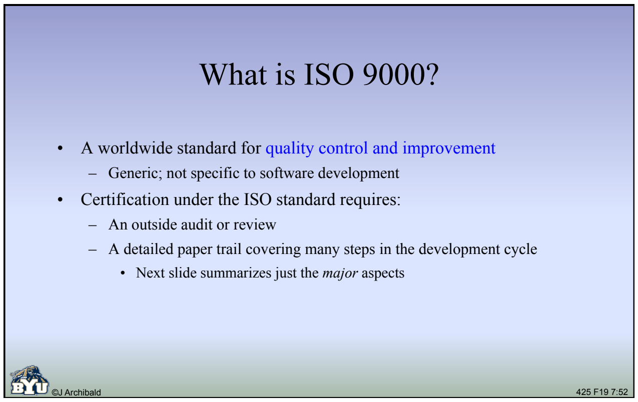
©J Archibald 425 F19 7:49


WE'RE HAVING AN ISO 9000 AUDIT THIS WEEK.
TAKE A LOOK AT YOUR DOCUMENTED JOB DESCRIPTIONS AND MAKE SURE THAT IT'S WHAT YOU'RE DOING IF THE AUDITOR ASKS.
ACCORDING TO THIS I'M SOME SORT OF ENGINEER.
AS IF WE'D HAVE TIME FOR THAT...
I'VE BEEN HIRED BY YOUR COMPANY TO PERFORM AN ISO 9000 AUDIT.
BASICALLY, YOU GIVE ME MONEY AND I TELL YOU THAT YOU'RE A BUNCH OF DOLTS. IT'S THE PERFECT JOB FOR ME.
TELL ME WHAT YOU DO HERE, ALICE, IF THAT'S YOUR REAL NAME.
I'M AN ENGINEER. I MAKE SLIDES THAT PEOPLE CAN'T READ. SOMETIMES I EAT DONUTS.

©J Archibald 425 F19 7:50


HERE ARE THE RESULTS OF MY ISO 9000 AUDIT OF YOUR COMPANY.
YOUR EMPLOYEES ARE LARGELY UNTRAINED AND — I COULDN'T HELP NOTICE — FAIRLY UNATTRACTIVE.
HOWEVER, THEY ARE ALSO HIGHLY SKILLED LIARS, SO YOU PASSED THE AUDIT EASILY.
WE SUCCEED WHERE IT COUNTS!!!

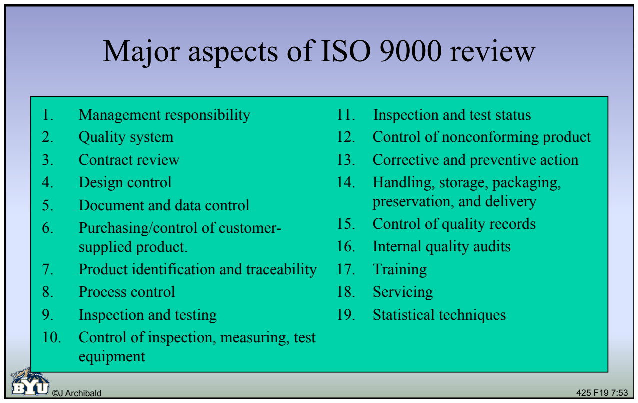
©J Archibald 425 F19 7:51



What is ISO 9000?

- A worldwide standard for quality control and improvement
  - Generic; not specific to software development
- Certification under the ISO standard requires:
  - An outside audit or review
  - A detailed paper trail covering many steps in the development cycle
    - Next slide summarizes just the *major* aspects

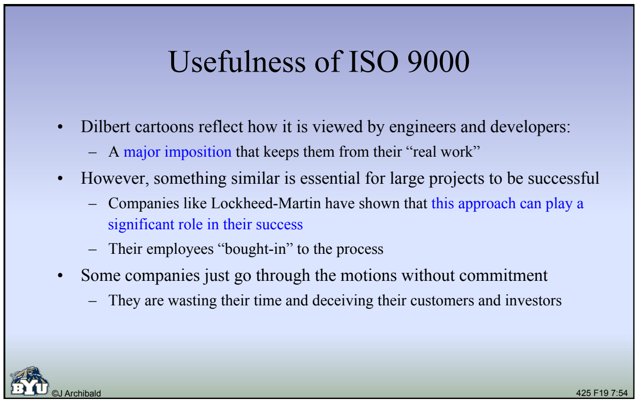
©J Archibald 425 F19 7:52



Major aspects of ISO 9000 review

1. Management responsibility
2. Quality system
3. Contract review
4. Design control
5. Document and data control
6. Purchasing/control of customer-supplied product.
7. Product identification and traceability
8. Process control
9. Inspection and testing
10. Control of inspection, measuring, test equipment
11. Inspection and test status
12. Control of nonconforming product
13. Corrective and preventive action
14. Handling, storage, packaging, preservation, and delivery
15. Control of quality records
16. Internal quality audits
17. Training
18. Servicing
19. Statistical techniques

©J Archibald 425 F19 7:53



Usefulness of ISO 9000

- Dilbert cartoons reflect how it is viewed by engineers and developers:
  - A major imposition that keeps them from their "real work"
- However, something similar is essential for large projects to be successful
  - Companies like Lockheed-Martin have shown that this approach can play a significant role in their success
  - Their employees "bought-in" to the process
- Some companies just go through the motions without commitment
  - They are wasting their time and deceiving their customers and investors

©J Archibald 425 F19 7:54

Capability Maturity Model

- Origin:
  - In early 1980s, US DoD became exasperated with delays and cost overruns in software projects by its contractors
  - Helped create the Software Engineering Institute to study ways to help the software industry grow responsibly
  - In 1987, SEI established the *software capability evaluation* (SCE)
    - A formal way to determine the maturity of an organization's software development process
    - A general measure of software development competence
  - CMM, introduced in 1991, ranks a potential contractor's software maturity from Level 1 to Level 5

Capability Maturity Model

- How it is used:
  - DoD releases RFP (request for proposal) for a project to be completed
    - RFP describes work to be done, contract terms, and minimum CMM ranking
  - Candidate must have undergone SCE, including site visit
    - Interviews of personnel, reviews of practices, observations of work environment
- Problems:
  - Different groups used different evaluation methodologies
  - Evaluation teams were staffed unevenly; some lacked experience
  - A thorough review is expensive (typical: tens of thousands of dollars)
  - Review completed in one week; tough to thoroughly address everything
  - Contractors learned to appear better than they really were

CMM levels

- January 2005: almost 2000 government and commercial organizations voluntarily reported CMM levels

53 % {
  - Level 1: Initial — Uncertainty
    - No organized quality activities
  - Level 2: Repeatable — Awakening
    - Short term motivational efforts

30 % {
  - Level 3: Defined — Enlightenment
    - Implementing 14 step program

17 % {
  - Level 4: Quantitatively managed — Wisdom
    - Continuing 14 step program
  - Level 5: Optimizing — Certainty
    - Quality improvement essential part of system

- Note that this is a self-selective group!
  - Companies with worst practices unlikely to subject themselves to a CMM evaluation

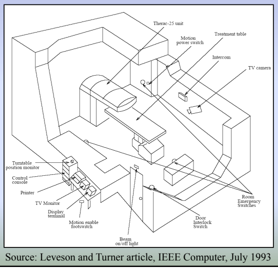
See Charette, Spectrum, Sept. 2005 ("Zen" labels from QMMG maturity matrix)

Customer confidence

- Certainly customers care about the quality of the product
- Since complex, embedded real-time systems are difficult to exhaustively test, the development process says a lot
  - Since customers cannot test the entire system, they have to believe that the design, development, and internal testing were thorough
  - Truly an exercise of faith!
  - Company's track-record on other projects is a key indicator
- Knowing that the development process was subject to rigorous internal scrutiny at every step helps to build customer confidence

Case study: how not to do design

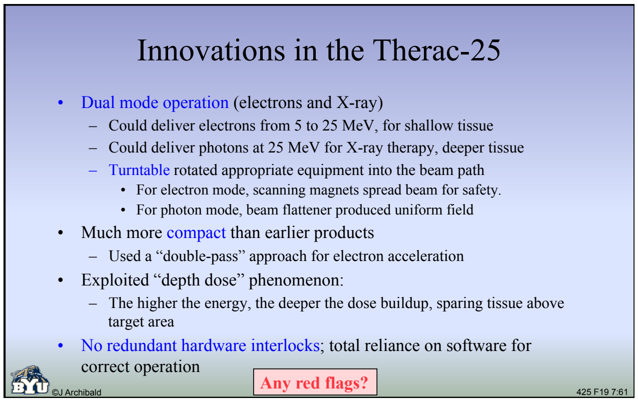
- The Therac-25, a medical linear accelerator; generated high-energy beams to destroy tumors with minimal impact on surrounding tissue



Source: Leveson and Turner article, IEEE Computer, July 1993

Corporate history

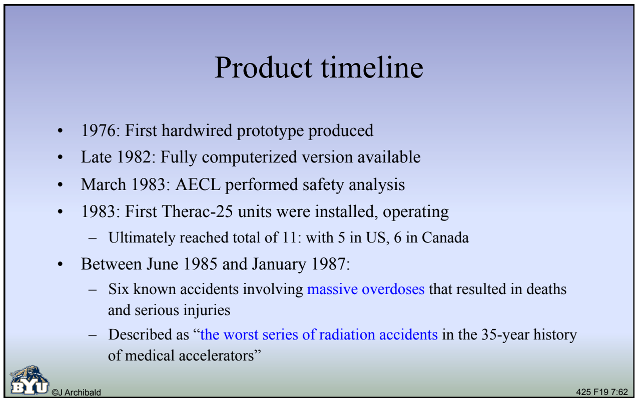
- Therac-25 was the product of a joint venture between Atomic Energy of Canada Limited (AECL) and CGR, a French company
- Earlier products from this joint venture:
  - Therac-6: a 6 million electron volt (MeV) accelerator for X-rays
  - Therac-20: a 20 MeV dual mode (X rays or electrons) accelerator
  - Characteristics of these machines:
    - Both based on older CGR machines, augmented with computerized control running on a DEC PDP-11 minicomputer
    - Both had limited software functionality: computer control merely added convenience in operating stand-alone hardware
    - Both included industry-standard safety features and hardware interlocks



Innovations in the Therac-25

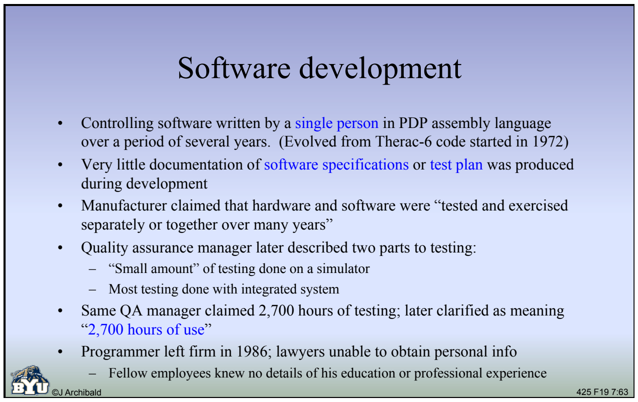
- Dual mode operation (electrons and X-ray)
  - Could deliver electrons from 5 to 25 MeV, for shallow tissue
  - Could deliver photons at 25 MeV for X-ray therapy, deeper tissue
  - Turntable rotated appropriate equipment into the beam path
    - For electron mode, scanning magnets spread beam for safety.
    - For photon mode, beam flattener produced uniform field
- Much more compact than earlier products
  - Used a "double-pass" approach for electron acceleration
- Exploited "depth dose" phenomenon:
  - The higher the energy, the deeper the dose buildup, sparing tissue above target area
- No redundant hardware interlocks; total reliance on software for correct operation

Any red flags?



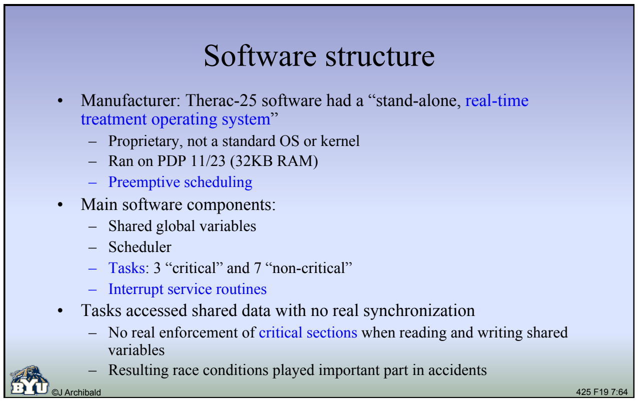
Product timeline

- 1976: First hardwired prototype produced
- Late 1982: Fully computerized version available
- March 1983: AECL performed safety analysis
- 1983: First Therac-25 units were installed, operating
  - Ultimately reached total of 11: with 5 in US, 6 in Canada
- Between June 1985 and January 1987:
  - Six known accidents involving massive overdoses that resulted in deaths and serious injuries
  - Described as "the worst series of radiation accidents in the 35-year history of medical accelerators"



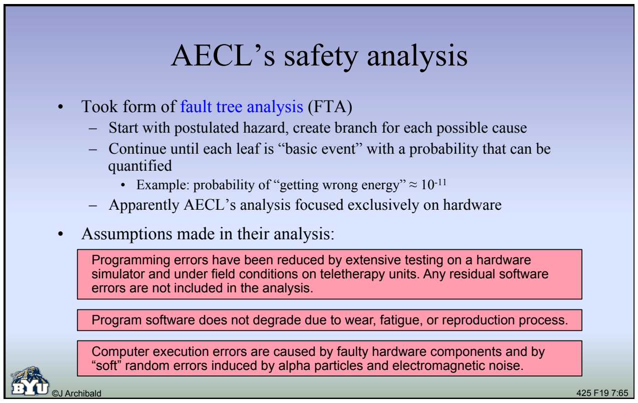
Software development

- Controlling software written by a single person in PDP assembly language over a period of several years. (Evolved from Therac-6 code started in 1972)
- Very little documentation of software specifications or test plan was produced during development
- Manufacturer claimed that hardware and software were "tested and exercised separately or together over many years"
- Quality assurance manager later described two parts to testing:
  - "Small amount" of testing done on a simulator
  - Most testing done with integrated system
- Same QA manager claimed 2,700 hours of testing; later clarified as meaning "2,700 hours of use"
- Programmer left firm in 1986; lawyers unable to obtain personal info
  - Fellow employees knew no details of his education or professional experience



Software structure

- Manufacturer: Therac-25 software had a "stand-alone, real-time treatment operating system"
  - Proprietary, not a standard OS or kernel
  - Ran on PDP 11/23 (32KB RAM)
  - Preemptive scheduling
- Main software components:
  - Shared global variables
  - Scheduler
  - Tasks: 3 "critical" and 7 "non-critical"
  - Interrupt service routines
- Tasks accessed shared data with no real synchronization
  - No real enforcement of critical sections when reading and writing shared variables
  - Resulting race conditions played important part in accidents



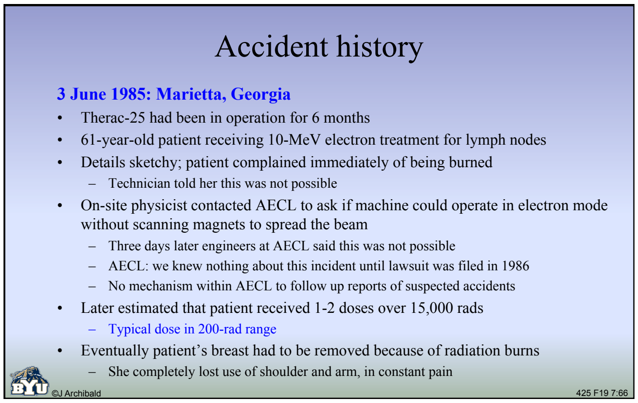
AECL's safety analysis

- Took form of fault tree analysis (FTA)
  - Start with postulated hazard, create branch for each possible cause
  - Continue until each leaf is "basic event" with a probability that can be quantified
    - Example: probability of "getting wrong energy" ≈ 10^{-11}
  - Apparently AECL's analysis focused exclusively on hardware
- Assumptions made in their analysis:

Programming errors have been reduced by extensive testing on a hardware simulator and under field conditions on teletherapy units. Any residual software errors are not included in the analysis.

Program software does not degrade due to wear, fatigue, or reproduction process.

Computer execution errors are caused by faulty hardware components and by "soft" random errors induced by alpha particles and electromagnetic noise.



Accident history

3 June 1985: Marietta, Georgia

- Therac-25 had been in operation for 6 months
- 61-year-old patient receiving 10-MeV electron treatment for lymph nodes
- Details sketchy; patient complained immediately of being burned
  - Technician told her this was not possible
- On-site physicist contacted AECL to ask if machine could operate in electron mode without scanning magnets to spread the beam
  - Three days later engineers at AECL said this was not possible
  - AECL: we knew nothing about this incident until lawsuit was filed in 1986
  - No mechanism within AECL to follow up reports of suspected accidents
- Later estimated that patient received 1-2 doses over 15,000 rads
  - Typical dose in 200-rad range
- Eventually patient's breast had to be removed because of radiation burns
  - She completely lost use of shoulder and arm, in constant pain

Accident history

26 July 1985: Hamilton, Ontario

- 5 seconds into treatment, machine shut down with error message, but display indicated “no dose”
  - Operator tried again with same result: machine shut down, “no dose” displayed
  - Cycle repeated five times: “standard operating procedure”
  - Not an unusual scenario according to experienced operators
  - Hospital service technician checked out machine, but found nothing wrong
- Patient complained of burning sensation in treatment area in hip
  - Patient died in November of cancer; autopsy noted that total hip replacement would have been required as a result of radiation overexposure
  - Technician later estimated that patient received 13,000 to 17,000 rads
- AECL could not reproduce problem; switches on turntable were blamed
  - Turntable operation was modified, including software that read switches
  - Customers were told that “analysis of the hazard rate of the new solution indicates an improvement over the old system by at least five orders of magnitude”
  - AECL later admitted that switch testing was “inconclusive”

Accident history

December 1985, Yakima, Washington

- After treatment, patient developed excessive reddening of the skin in a parallel striped pattern on her right hip
  - Staff could not find explanation that made sense; could not reproduce hardware arrangement with matching orientation of stripes
  - AECL was informed via letter and phone calls
- Written response from AECL:
  - “After careful consideration, we are of the opinion that this damage could not have been produced by any malfunction of the Therac-25 or by any operator error.”
  - Included two pages of technical reasons why an overdose was impossible
  - Stated that there had been “apparently no other instances of similar damage to this or other patients”
- Machine malfunction not acknowledged until later accidents understood

Accident history

March 1986, Tyler, Texas

- More details known because of diligence of hospital physicist
- Experienced operator entered prescription data, noticed an error
  - She had typed ‘x’ (for X-ray) when ‘e’ (for electron) was intended
  - Used cursor-up key to edit the mode entry, then hit return several times to move to bottom of screen
- After message from computer that parameters had been verified, she began treatment
- Console displayed message “Malfunction 54”
  - Only on-site information (sheet on side of machine) indicated that this was a “dose input 2” error; no other information available
  - Undocumented meaning: delivered dose was either too high or too low
- Machine showed substantial underdose on dose monitor display

Tyler accident, cont.

- Operator repeated treatment, got same message
  - Operator was isolated from patient; machine in shielded room
  - Video monitor was unplugged, audio monitor was broken
- Patient felt electric shock, burning as if hot coffee poured on his back
  - Not his first treatment, knew this was not normal
  - Started to get up to get help just as second treatment began
  - Felt shock in arm, as though his hand were leaving his body
  - Went to door and pounded on it to surprise of operator
- Electrical shock assumed initially; the machine was shut down for testing
  - Full day of testing could not reproduce “Malfunction 54” message
  - AECL engineer maintained that overdose with machine was impossible
  - AECL told physicist: no prior accidents involving radiation overexposure
  - Independent engineers concluded that machine could not shock patient
- Patient died five months later from complications of overdose
  - Estimated to have received 16,500 to 25,000 rads in small area

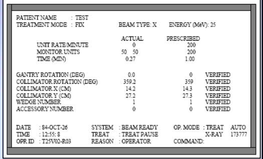
Accident history

April 1986, Tyler, Texas (same facility, 1 month later)

- Operator entered prescription data, noticed error
  - Used cursor-up key to change from X-ray to electron; continued with treatment
  - After a few seconds, machine shut down with loud noise (intercom was working)
  - Console displayed “Malfunction 54”
  - Operator rushed in, patient moaned for help, said he felt “fire” on side of his face
  - Operator got physicist immediately
- Patient died three weeks later
  - Autopsy showed high-dose radiation injury to brain
- Physicist worked at length with operator to reproduce error
  - Eventually produced error message at will, then tried to measure actual dosage
  - With his help, AECL was finally able to reproduce malfunction on their machine
  - AECL measured dosage at center of field at 25,000 rads
  - Critical factor: data entry speed during editing

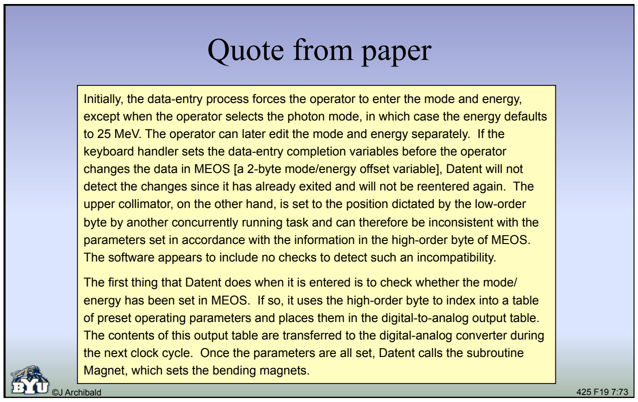
The bug

- Task that handled data entry relied on separate keyboard handler task to get input from operator
  - Communication between the tasks used “data entry completion flag” to determine if prescription data had been entered
  - Code structure, race conditions on flag allowed data entry task to completely miss editing changes in already entered data
  - Editing changes were displayed on operator screen and internal variable was actually changed, but machine control routine would use old value
  - Software did not perform consistency check
- Fundamental problem was difficult to see; full software was complex



```
PATIENT NAME      : TEST
TREATMENT MODE    : FIX              BEAM TYPE: X      ENERGY (MeV): 25

                                      ACTUAL          PRESCRIBED
          UNIT RATE/MINUTE               0               200
          MONITOR UNITS               50   50            200
          TIME (MIN)                    0.27             1.00

GANTRY ROTATION (DEG)                   0.0               0      VERIFIED
COLLIMATOR ROTATION (DEG)             359.2             359      VERIFIED
COLLIMATOR X (CM)                      14.2             14.3     VERIFIED
COLLIMATOR Y (CM)                      27.2             27.3     VERIFIED
WEDGE NUMBER                             1                1      VERIFIED
ACCESSORY NUMBER                         0                0      VERIFIED

DATE    : 84-OCT-26      SYSTEM   : BEAM READY     OP. MODE  : TREAT    AUTO
TIME    : 12:55: 8       TREAT    : TREAT PAUSE                X-RAY    173777
OPR ID  : T25V02-R03     REASON   : OPERATOR       COMMAND:
```

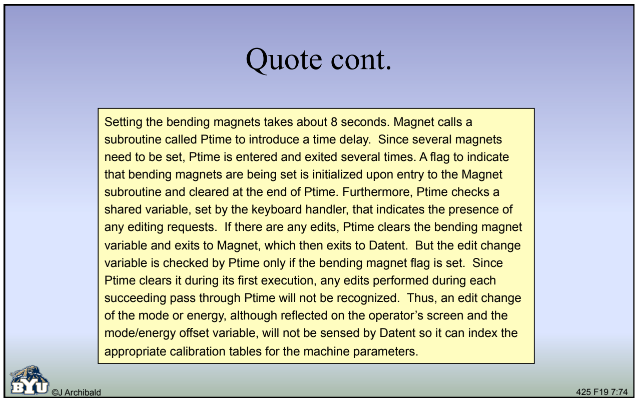
Operator screen



Quote from paper

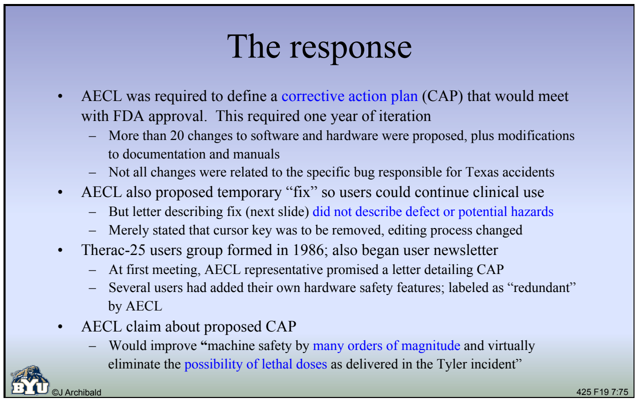
Initially, the data-entry process forces the operator to enter the mode and energy, except when the operator selects the photon mode, in which case the energy defaults to 25 MeV. The operator can later edit the mode and energy separately. If the keyboard handler sets the data-entry completion variables before the operator changes the data in MEOS [a 2-byte mode/energy offset variable], Datent will not detect the changes since it has already exited and will not be reentered again. The upper collimator, on the other hand, is set to the position dictated by the low-order byte by another concurrently running task and can therefore be inconsistent with the parameters set in accordance with the information in the high-order byte of MEOS. The software appears to include no checks to detect such an incompatibility.

The first thing that Datent does when it is entered is to check whether the mode/energy has been set in MEOS. If so, it uses the high-order byte to index into a table of preset operating parameters and places them in the digital-to-analog output table. The contents of this output table are transferred to the digital-analog converter during the next clock cycle. Once the parameters are all set, Datent calls the subroutine Magnet, which sets the bending magnets.



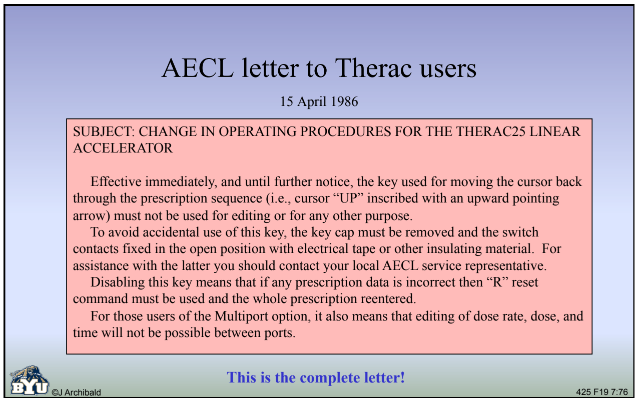
Quote cont.

Setting the bending magnets takes about 8 seconds. Magnet calls a subroutine called Ptime to introduce a time delay. Since several magnets need to be set, Ptime is entered and exited several times. A flag to indicate that bending magnets are being set is initialized upon entry to the Magnet subroutine and cleared at the end of Ptime. Furthermore, Ptime checks a shared variable, set by the keyboard handler, that indicates the presence of any editing requests. If there are any edits, Ptime clears the bending magnet variable and exits to Magnet, which then exits to Datent. But the edit change variable is checked by Ptime only if the bending magnet flag is set. Since Ptime clears it during its first execution, any edits performed during each succeeding pass through Ptime will not be recognized. Thus, an edit change of the mode or energy, although reflected on the operator's screen and the mode/energy offset variable, will not be sensed by Datent so it can index the appropriate calibration tables for the machine parameters.



The response

- AECL was required to define a corrective action plan (CAP) that would meet with FDA approval. This required one year of iteration
  - More than 20 changes to software and hardware were proposed, plus modifications to documentation and manuals
  - Not all changes were related to the specific bug responsible for Texas accidents
- AECL also proposed temporary "fix" so users could continue clinical use
  - But letter describing fix (next slide) did not describe defect or potential hazards
  - Merely stated that cursor key was to be removed, editing process changed
- Therac-25 users group formed in 1986; also began user newsletter
  - At first meeting, AECL representative promised a letter detailing CAP
  - Several users had added their own hardware safety features; labeled as "redundant" by AECL
- AECL claim about proposed CAP
  - Would improve "machine safety by many orders of magnitude and virtually eliminate the possibility of lethal doses as delivered in the Tyler incident"



AECL letter to Therac users

15 April 1986

SUBJECT: CHANGE IN OPERATING PROCEDURES FOR THE THERAC25 LINEAR ACCELERATOR

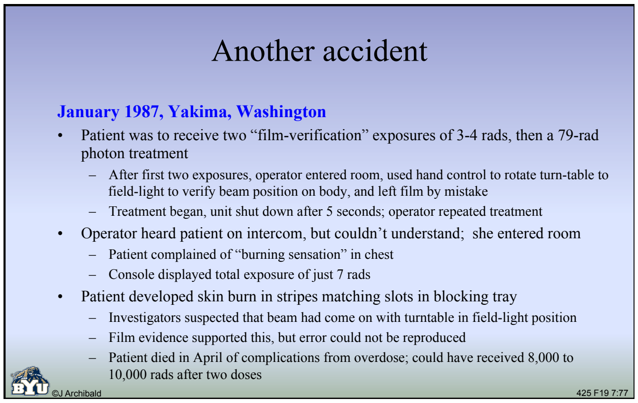
Effective immediately, and until further notice, the key used for moving the cursor back through the prescription sequence (i.e., cursor "UP" inscribed with an upward pointing arrow) must not be used for editing or for any other purpose.

To avoid accidental use of this key, the key cap must be removed and the switch contacts fixed in the open position with electrical tape or other insulating material. For assistance with the latter you should contact your local AECL service representative.

Disabling this key means that if any prescription data is incorrect then "R" reset command must be used and the whole prescription reentered.

For those users of the Multiport option, it also means that editing of dose rate, dose, and time will not be possible between ports.

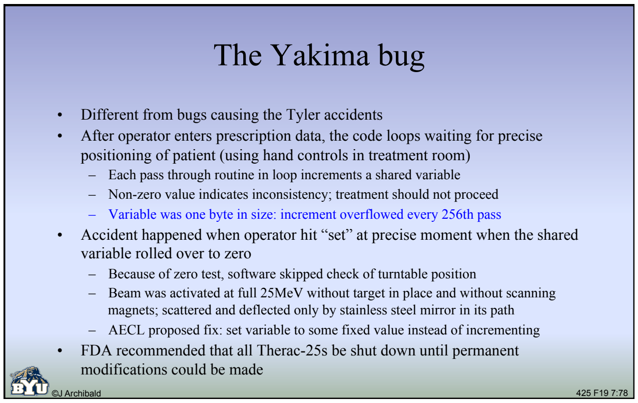
This is the complete letter!



Another accident

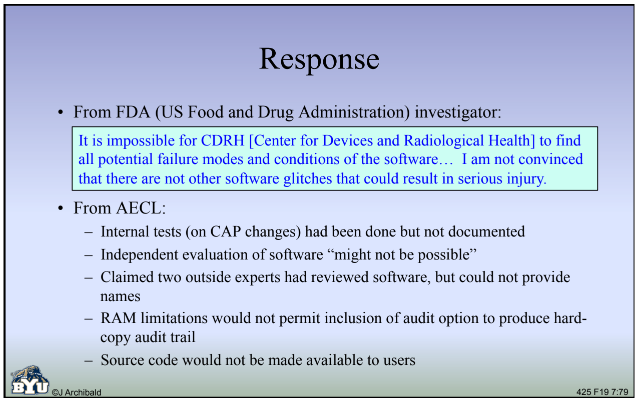
January 1987, Yakima, Washington

- Patient was to receive two "film-verification" exposures of 3-4 rads, then a 79-rad photon treatment
  - After first two exposures, operator entered room, used hand control to rotate turn-table to field-light to verify beam position on body, and left film by mistake
  - Treatment began, unit shut down after 5 seconds; operator repeated treatment
- Operator heard patient on intercom, but couldn't understand; she entered room
  - Patient complained of "burning sensation" in chest
  - Console displayed total exposure of just 7 rads
- Patient developed skin burn in stripes matching slots in blocking tray
  - Investigators suspected that beam had come on with turntable in field-light position
  - Film evidence supported this, but error could not be reproduced
  - Patient died in April of complications from overdose; could have received 8,000 to 10,000 rads after two doses



The Yakima bug

- Different from bugs causing the Tyler accidents
- After operator enters prescription data, the code loops waiting for precise positioning of patient (using hand controls in treatment room)
  - Each pass through routine in loop increments a shared variable
  - Non-zero value indicates inconsistency; treatment should not proceed
  - Variable was one byte in size: increment overflowed every 256th pass
- Accident happened when operator hit "set" at precise moment when the shared variable rolled over to zero
  - Because of zero test, software skipped check of turntable position
  - Beam was activated at full 25MeV without target in place and without scanning magnets; scattered and deflected only by stainless steel mirror in its path
  - AECL proposed fix: set variable to some fixed value instead of incrementing
- FDA recommended that all Therac-25s be shut down until permanent modifications could be made

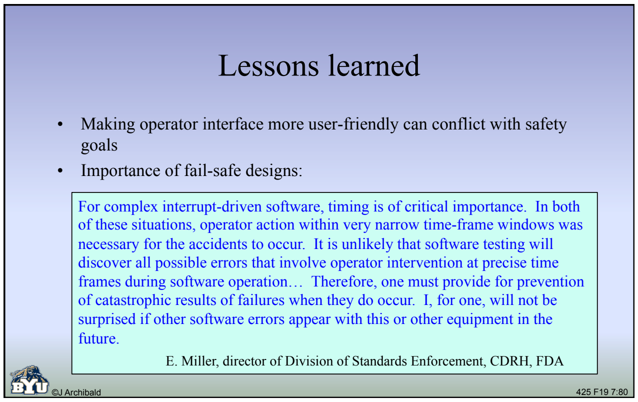


Response

- From FDA (US Food and Drug Administration) investigator:

> It is impossible for CDRH [Center for Devices and Radiological Health] to find all potential failure modes and conditions of the software… I am not convinced that there are not other software glitches that could result in serious injury.

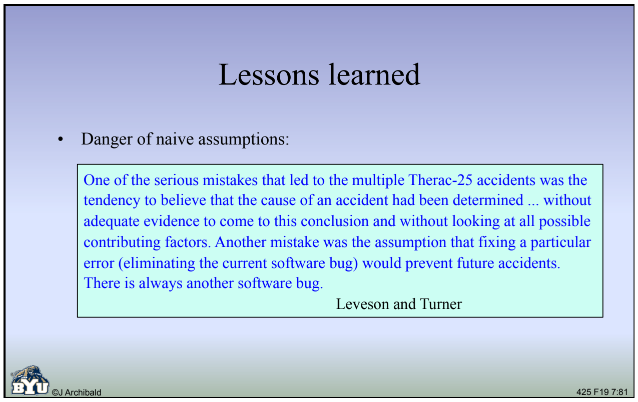
- From AECL:
  - Internal tests (on CAP changes) had been done but not documented
  - Independent evaluation of software "might not be possible"
  - Claimed two outside experts had reviewed software, but could not provide names
  - RAM limitations would not permit inclusion of audit option to produce hard-copy audit trail
  - Source code would not be made available to users



Lessons learned

- Making operator interface more user-friendly can conflict with safety goals
- Importance of fail-safe designs:

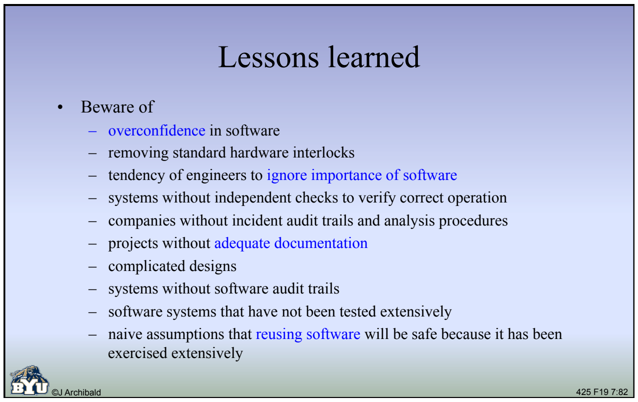
> For complex interrupt-driven software, timing is of critical importance. In both of these situations, operator action within very narrow time-frame windows was necessary for the accidents to occur. It is unlikely that software testing will discover all possible errors that involve operator intervention at precise time frames during software operation… Therefore, one must provide for prevention of catastrophic results of failures when they do occur. I, for one, will not be surprised if other software errors appear with this or other equipment in the future.
>
> E. Miller, director of Division of Standards Enforcement, CDRH, FDA



Lessons learned

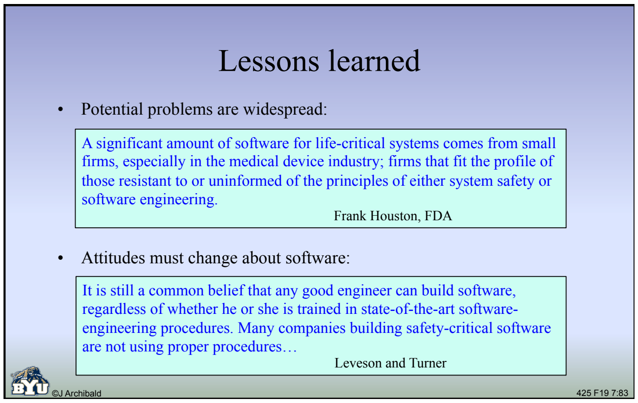
- Danger of naive assumptions:

> One of the serious mistakes that led to the multiple Therac-25 accidents was the tendency to believe that the cause of an accident had been determined ... without adequate evidence to come to this conclusion and without looking at all possible contributing factors. Another mistake was the assumption that fixing a particular error (eliminating the current software bug) would prevent future accidents. There is always another software bug.
>
> Leveson and Turner



Lessons learned

- Beware of
  - overconfidence in software
  - removing standard hardware interlocks
  - tendency of engineers to ignore importance of software
  - systems without independent checks to verify correct operation
  - companies without incident audit trails and analysis procedures
  - projects without adequate documentation
  - complicated designs
  - systems without software audit trails
  - software systems that have not been tested extensively
  - naive assumptions that reusing software will be safe because it has been exercised extensively



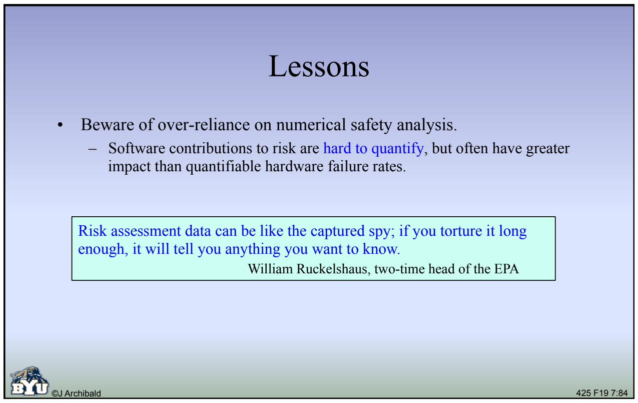
Lessons learned

- Potential problems are widespread:

> A significant amount of software for life-critical systems comes from small firms, especially in the medical device industry; firms that fit the profile of those resistant to or uninformed of the principles of either system safety or software engineering.
>
> Frank Houston, FDA

- Attitudes must change about software:

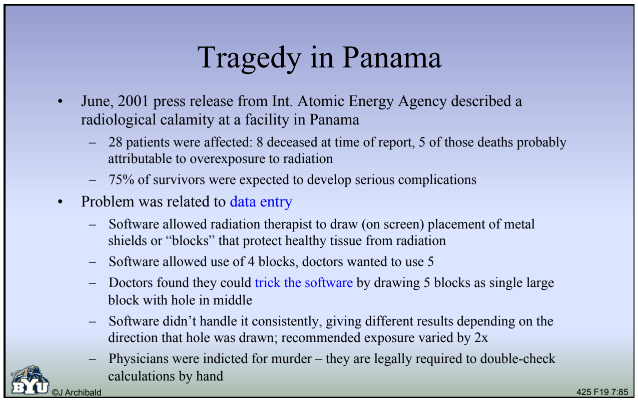
> It is still a common belief that any good engineer can build software, regardless of whether he or she is trained in state-of-the-art software-engineering procedures. Many companies building safety-critical software are not using proper procedures…
>
> Leveson and Turner



Lessons

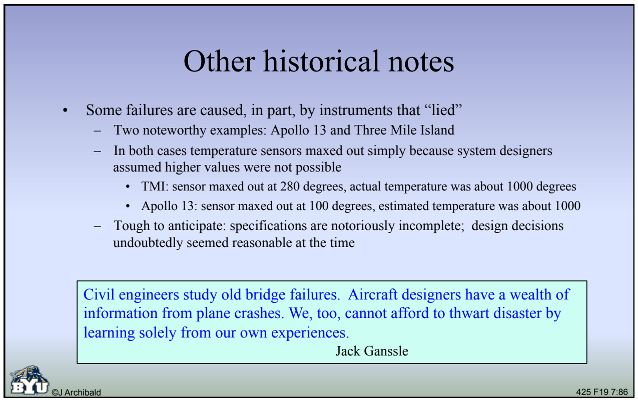
- Beware of over-reliance on numerical safety analysis.
  - Software contributions to risk are hard to quantify, but often have greater impact than quantifiable hardware failure rates.

> Risk assessment data can be like the captured spy; if you torture it long enough, it will tell you anything you want to know.
>
> William Ruckelshaus, two-time head of the EPA



Tragedy in Panama

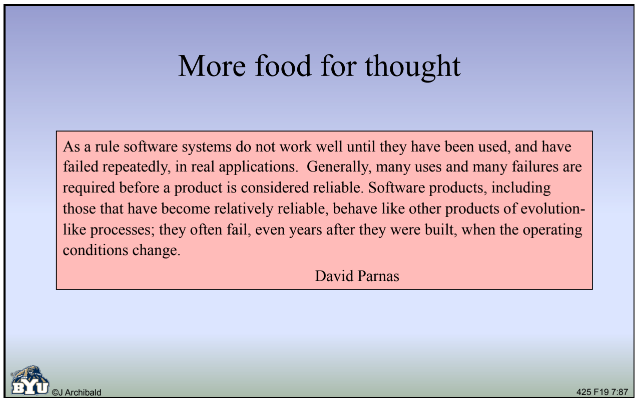
- June, 2001 press release from Int. Atomic Energy Agency described a radiological calamity at a facility in Panama
  - 28 patients were affected: 8 deceased at time of report, 5 of those deaths probably attributable to overexposure to radiation
  - 75% of survivors were expected to develop serious complications
- Problem was related to data entry
  - Software allowed radiation therapist to draw (on screen) placement of metal shields or "blocks" that protect healthy tissue from radiation
  - Software allowed use of 4 blocks, doctors wanted to use 5
  - Doctors found they could trick the software by drawing 5 blocks as single large block with hole in middle
  - Software didn't handle it consistently, giving different results depending on the direction that hole was drawn; recommended exposure varied by 2x
  - Physicians were indicted for murder – they are legally required to double-check calculations by hand



Other historical notes

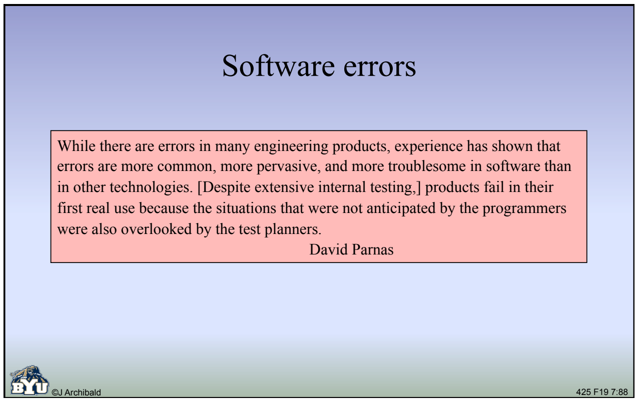
- Some failures are caused, in part, by instruments that "lied"
  - Two noteworthy examples: Apollo 13 and Three Mile Island
  - In both cases temperature sensors maxed out simply because system designers assumed higher values were not possible
    - TMI: sensor maxed out at 280 degrees, actual temperature was about 1000 degrees
    - Apollo 13: sensor maxed out at 100 degrees, estimated temperature was about 1000
  - Tough to anticipate: specifications are notoriously incomplete; design decisions undoubtedly seemed reasonable at the time

> Civil engineers study old bridge failures. Aircraft designers have a wealth of information from plane crashes. We, too, cannot afford to thwart disaster by learning solely from our own experiences.
>
> Jack Ganssle



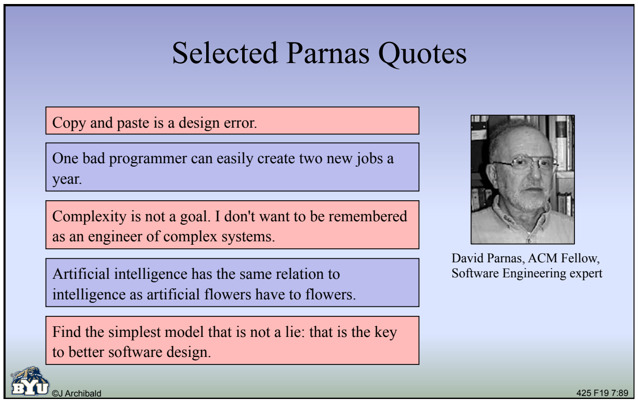
More food for thought

> As a rule software systems do not work well until they have been used, and have failed repeatedly, in real applications. Generally, many uses and many failures are required before a product is considered reliable. Software products, including those that have become relatively reliable, behave like other products of evolution-like processes; they often fail, even years after they were built, when the operating conditions change.
>
> David Parnas



Software errors

> While there are errors in many engineering products, experience has shown that errors are more common, more pervasive, and more troublesome in software than in other technologies. [Despite extensive internal testing,] products fail in their first real use because the situations that were not anticipated by the programmers were also overlooked by the test planners.
>
> David Parnas



Selected Parnas Quotes

Copy and paste is a design error.

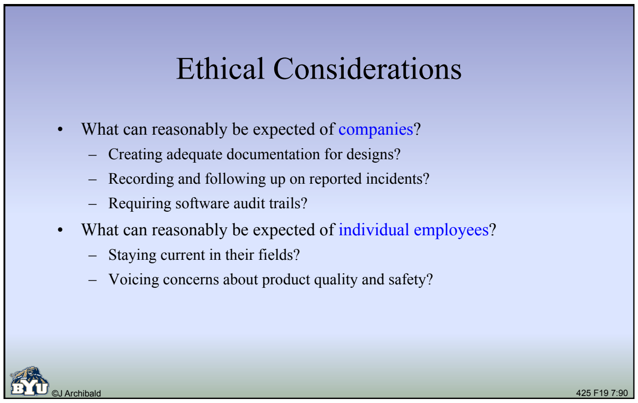
One bad programmer can easily create two new jobs a year.

Complexity is not a goal. I don't want to be remembered as an engineer of complex systems.

Artificial intelligence has the same relation to intelligence as artificial flowers have to flowers.

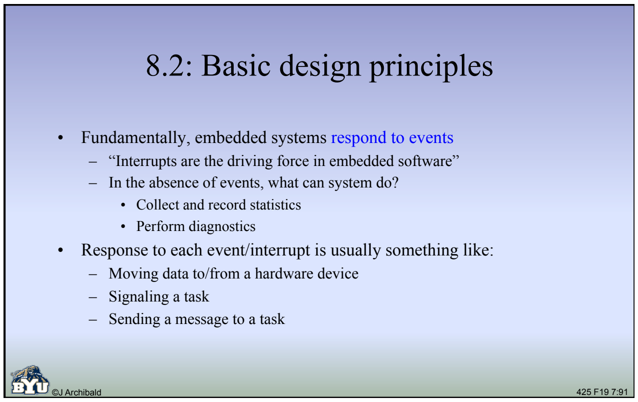
Find the simplest model that is not a lie: that is the key to better software design.

David Parnas, ACM Fellow, Software Engineering expert



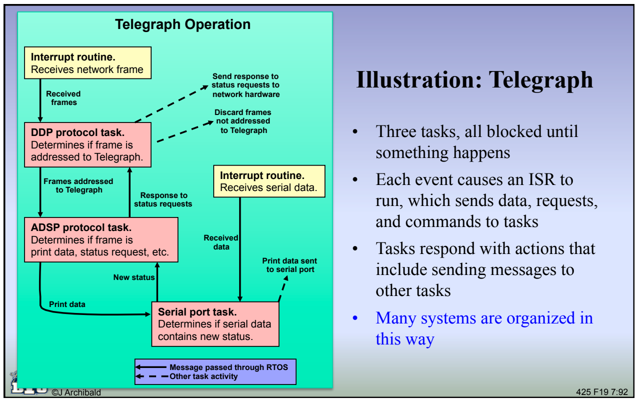
Ethical Considerations

- What can reasonably be expected of companies?
  - Creating adequate documentation for designs?
  - Recording and following up on reported incidents?
  - Requiring software audit trails?
- What can reasonably be expected of individual employees?
  - Staying current in their fields?
  - Voicing concerns about product quality and safety?



8.2: Basic design principles

- Fundamentally, embedded systems respond to events
  - "Interrupts are the driving force in embedded software"
  - In the absence of events, what can system do?
    - Collect and record statistics
    - Perform diagnostics
- Response to each event/interrupt is usually something like:
  - Moving data to/from a hardware device
  - Signaling a task
  - Sending a message to a task



Telegraph Operation

Interrupt routine.
Receives network frame

Received frames

Send response to status requests to network hardware

Discard frames not addressed to Telegraph

DDP protocol task.
Determines if frame is addressed to Telegraph.

Frames addressed to Telegraph

Response to status requests

Interrupt routine.
Receives serial data.

ADSP protocol task.
Determines if frame is print data, status request, etc.

Received data

Print data sent to serial port

New status

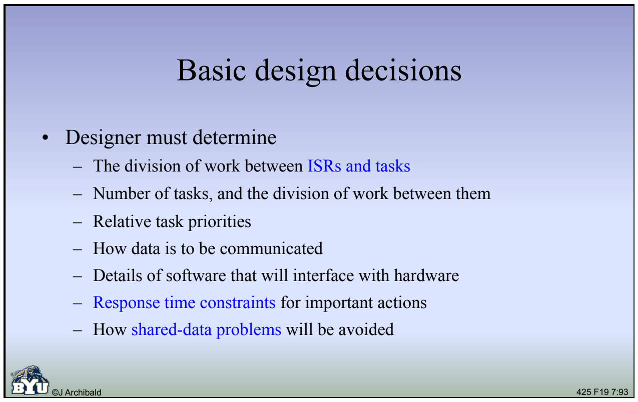
Print data

Serial port task.
Determines if serial data contains new status.

Message passed through RTOS
Other task activity

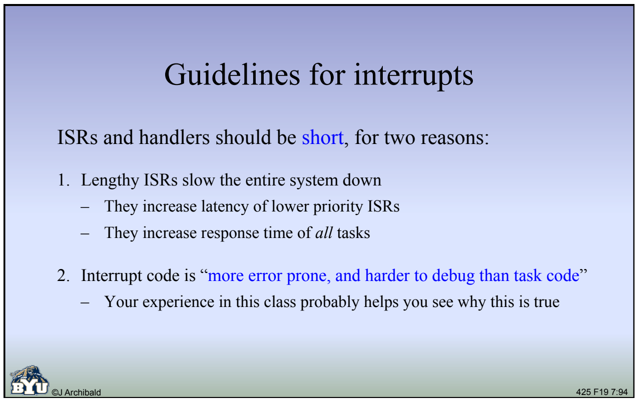
Illustration: Telegraph

- Three tasks, all blocked until something happens
- Each event causes an ISR to run, which sends data, requests, and commands to tasks
- Tasks respond with actions that include sending messages to other tasks
- Many systems are organized in this way



Basic design decisions

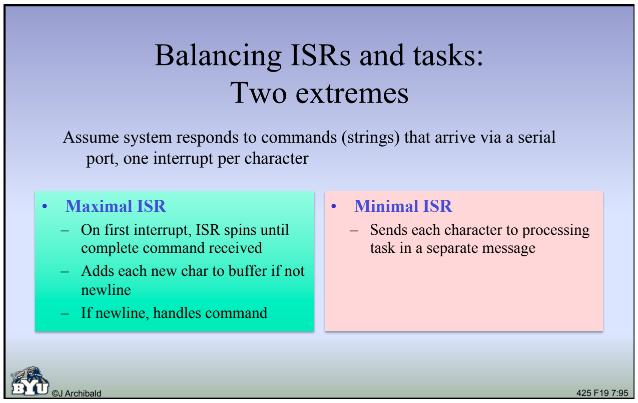
- Designer must determine
  - The division of work between ISRs and tasks
  - Number of tasks, and the division of work between them
  - Relative task priorities
  - How data is to be communicated
  - Details of software that will interface with hardware
  - Response time constraints for important actions
  - How shared-data problems will be avoided



Guidelines for interrupts

ISRs and handlers should be short, for two reasons:

1. Lengthy ISRs slow the entire system down
   - They increase latency of lower priority ISRs
   - They increase response time of *all* tasks
2. Interrupt code is "more error prone, and harder to debug than task code"
   - Your experience in this class probably helps you see why this is true

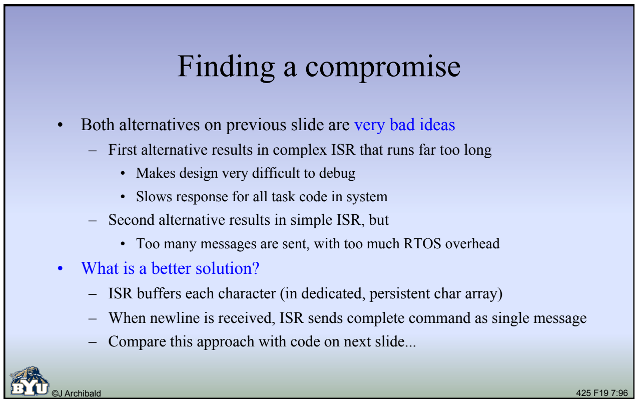


Balancing ISRs and tasks: Two extremes

Assume system responds to commands (strings) that arrive via a serial port, one interrupt per character

- **Maximal ISR**
  - On first interrupt, ISR spins until complete command received
  - Adds each new char to buffer if not newline
  - If newline, handles command

- **Minimal ISR**
  - Sends each character to processing task in a separate message



Finding a compromise

- Both alternatives on previous slide are very bad ideas
  - First alternative results in complex ISR that runs far too long
    - Makes design very difficult to debug
    - Slows response for all task code in system
  - Second alternative results in simple ISR, but
    - Too many messages are sent, with too much RTOS overhead
- What is a better solution?
  - ISR buffers each character (in dedicated, persistent char array)
  - When newline is received, ISR sends complete command as single message
  - Compare this approach with code on next slide...

```
#define SIZEOF_CMD_BUFFER 200
char a_chCommandBuffer[SIZEOF_CMD_BUFFER];
#define MSG_EMPTY ((char *) 0)
char *mboxCommand = MSG_EMPTY;
#define MSG_COMMAND_ARRIVED ((char *) 1)

void interrupt vGetCommandCharacter (void)
{
   static char *p_chCommandBufferTail = a_chCommandBuffer;
   int iError;

   *p_chCommandBufferTail = !! char read from hardware
   if (*p_chCommandBufferTail == '\r')
      sc_post (&mboxCommand, MSG_COMMAND_ARRIVED, &iError);

   /* Advance the tail pointer and wrap if necessary */
   ++p_chCommandBufferTail;
   if (p_chCommandBufferTail ==  &a_chCommandBuffer[SIZEOF_CMD_BUFFER])
      p_chCommandBufferTail = a_chCommandBuffer;

   !! Reset the hardware as necessary
}

void vInterpretCommandTask (void)
{
   static char *p_chCommandBufferHead = a_chCommandBuffer;
   int iError;

   while (TRUE) {
      /* wait for next command to arrive */
      sc_pend (&mboxCommand, WAIT_FOREVER, &iError);

      /* we have a command */
      !! interpret the command received
      !! advance p_chCommandBufferHead past <cr>
}
```

Fig. 8.2

Is this well-written code?

Command buffer is a shared data structure accessed using head and tail pointers that work on different parts of the array, so we shouldn't have a shared data problem.

But, message is just a flag saying the next command is available. A semaphore would be a better choice here.

Moreover, the code does not test for overflow, either in the command buffer or in the mailbox.

How many tasks?

- Advantages of having many tasks:
  - Tasks will be smaller, simpler, easier to write and debug
  - Tasks can be dedicated to servicing a single event; a separate task can be used for each type of event
  - Easier to make task code modular
  - Often easier to encapsulate data and hardware details within tasks
  - Designer has more control over relative response times for much of the work performed by tasks

How many tasks?

- Disadvantages of having many tasks:
  - More memory needed for stacks, message buffers
  - More CPU time spent switching tasks
  - More calls to RTOS, increased system overhead
  - More likely to have data sharing between tasks, increasing the likelihood of shared-data problems
  - More need for semaphores to protect shared data, increasing overhead and the likelihood of semaphore bugs
  - More need for messages, queues, pipes for communication between tasks, increasing overhead and the likelihood of related bugs

Comparing the tradeoffs

- The playing field is not level:
  - If you have many tasks, the negative consequences are *automatic*
  - The advantages of many tasks come *only* if you use tasks well, and if your design does a good job of dividing the work
- The bottom-line recommendation:

  "Other things being equal, use as few tasks as you can get away with; add more tasks to your design only for clear reasons."

General suggestions

- All else being equal:
  - Have small, simple tasks
  - Have a separate task for work done in response to each different event
- Example system (printer) at right:
  - One task handles printer's paper mechanism
  - One task handles button presses on front panel
  - One task handles display: resolves conflicts between queued messages from other tasks

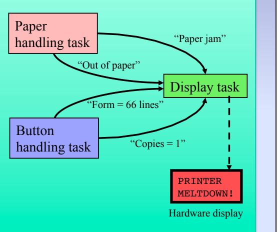


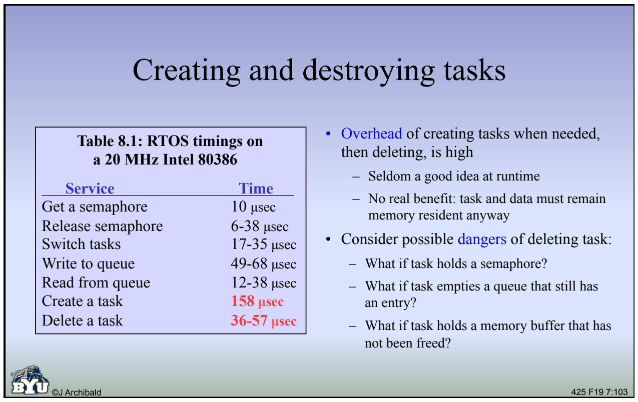
Figure 8.3

```
/* vtaska.c */
!! Private static data is declared here

void vTaskA (void)
{
   !! More private data declared here,
   !! either static or on the stack

   !! Initialization code, if needed

   while (FOREVER)
   {
      !! Wait for system signal (event, msg, etc.)

      switch (!! type of signal)
      {
         case !! signal type 1:
            …
            break;

         case !! signal type 2:
            …
            break;
         …
      }
   }
}
```

Fig. 8.5

Recommended task structure

- Spin in loop, waiting for signal or message
- Block in only one place; behavior is easier to understand
- Task response time is predictable, more easily determined

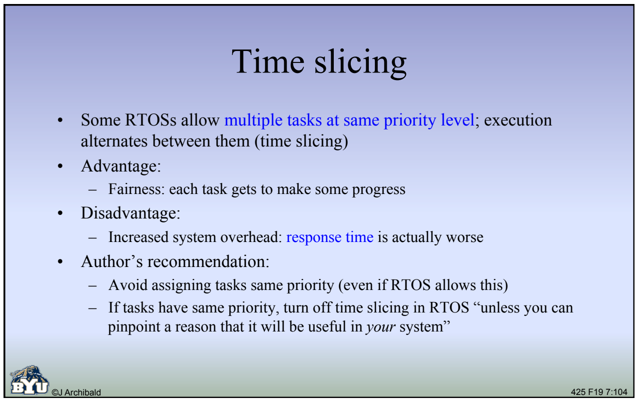


Creating and destroying tasks

Table 8.1: RTOS timings on a 20 MHz Intel 80386

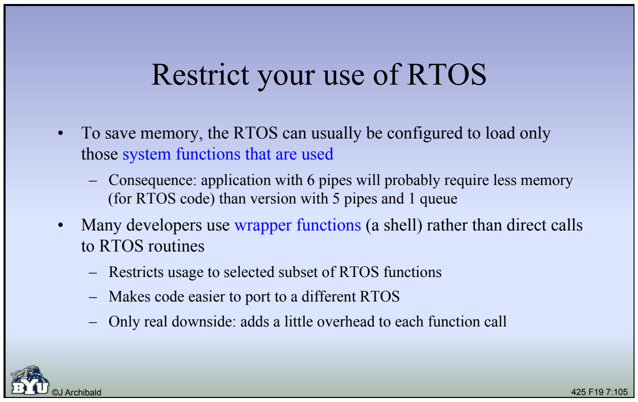
Service	Time
Get a semaphore	10 µsec
Release semaphore	6-38 µsec
Switch tasks	17-35 µsec
Write to queue	49-68 µsec
Read from queue	12-38 µsec
Create a task	158 µsec
Delete a task	36-57 µsec

- Overhead of creating tasks when needed, then deleting, is high
  - Seldom a good idea at runtime
  - No real benefit: task and data must remain memory resident anyway
- Consider possible dangers of deleting task:
  - What if task holds a semaphore?
  - What if task empties a queue that still has an entry?
  - What if task holds a memory buffer that has not been freed?



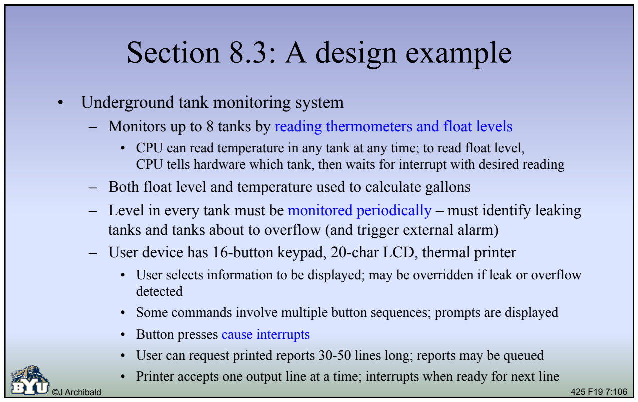
Time slicing

- Some RTOSs allow multiple tasks at same priority level; execution alternates between them (time slicing)
- Advantage:
  - Fairness: each task gets to make some progress
- Disadvantage:
  - Increased system overhead: response time is actually worse
- Author’s recommendation:
  - Avoid assigning tasks same priority (even if RTOS allows this)
  - If tasks have same priority, turn off time slicing in RTOS “unless you can pinpoint a reason that it will be useful in *your* system”



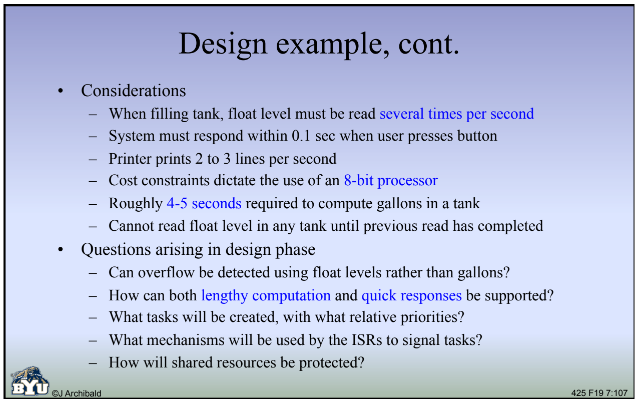
Restrict your use of RTOS

- To save memory, the RTOS can usually be configured to load only those system functions that are used
  - Consequence: application with 6 pipes will probably require less memory (for RTOS code) than version with 5 pipes and 1 queue
- Many developers use wrapper functions (a shell) rather than direct calls to RTOS routines
  - Restricts usage to selected subset of RTOS functions
  - Makes code easier to port to a different RTOS
  - Only real downside: adds a little overhead to each function call



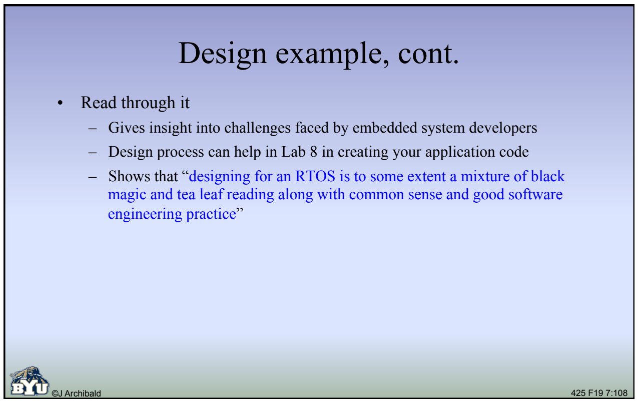
Section 8.3: A design example

- Underground tank monitoring system
  - Monitors up to 8 tanks by reading thermometers and float levels
    - CPU can read temperature in any tank at any time; to read float level, CPU tells hardware which tank, then waits for interrupt with desired reading
  - Both float level and temperature used to calculate gallons
  - Level in every tank must be monitored periodically – must identify leaking tanks and tanks about to overflow (and trigger external alarm)
  - User device has 16-button keypad, 20-char LCD, thermal printer
    - User selects information to be displayed; may be overridden if leak or overflow detected
    - Some commands involve multiple button sequences; prompts are displayed
    - Button presses cause interrupts
    - User can request printed reports 30-50 lines long; reports may be queued
    - Printer accepts one output line at a time; interrupts when ready for next line



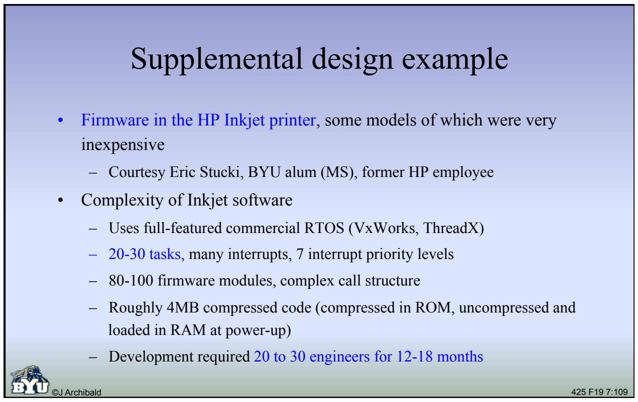
Design example, cont.

- Considerations
  - When filling tank, float level must be read several times per second
  - System must respond within 0.1 sec when user presses button
  - Printer prints 2 to 3 lines per second
  - Cost constraints dictate the use of an 8-bit processor
  - Roughly 4-5 seconds required to compute gallons in a tank
  - Cannot read float level in any tank until previous read has completed
- Questions arising in design phase
  - Can overflow be detected using float levels rather than gallons?
  - How can both lengthy computation and quick responses be supported?
  - What tasks will be created, with what relative priorities?
  - What mechanisms will be used by the ISRs to signal tasks?
  - How will shared resources be protected?



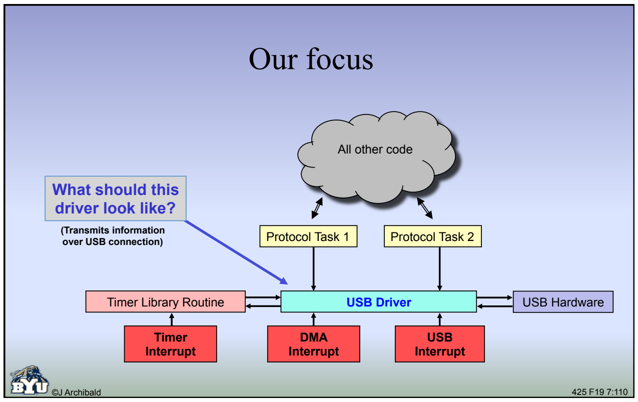
Design example, cont.

- Read through it
  - Gives insight into challenges faced by embedded system developers
  - Design process can help in Lab 8 in creating your application code
  - Shows that “designing for an RTOS is to some extent a mixture of black magic and tea leaf reading along with common sense and good software engineering practice”

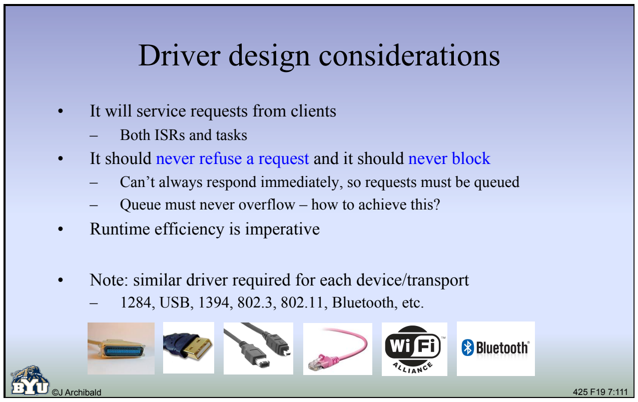


Supplemental design example

- Firmware in the HP Inkjet printer, some models of which were very inexpensive
  - Courtesy Eric Stucki, BYU alum (MS), former HP employee
- Complexity of Inkjet software
  - Uses full-featured commercial RTOS (VxWorks, ThreadX)
  - 20-30 tasks, many interrupts, 7 interrupt priority levels
  - 80-100 firmware modules, complex call structure
  - Roughly 4MB compressed code (compressed in ROM, uncompressed and loaded in RAM at power-up)
  - Development required 20 to 30 engineers for 12-18 months

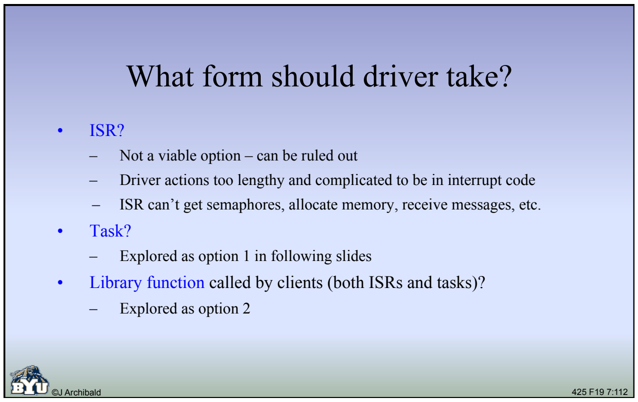


Our focus



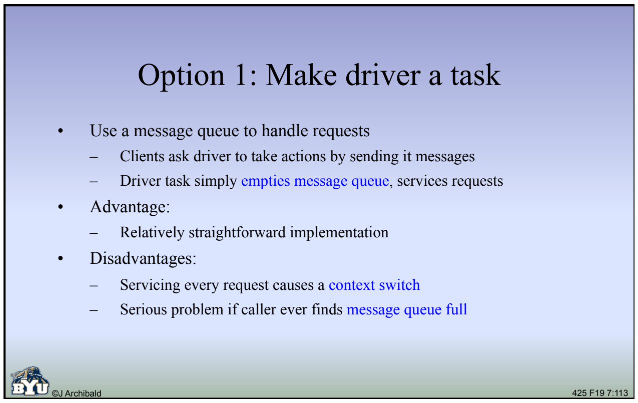
Driver design considerations

- It will service requests from clients
  - Both ISRs and tasks
- It should never refuse a request and it should never block
  - Can't always respond immediately, so requests must be queued
  - Queue must never overflow – how to achieve this?
- Runtime efficiency is imperative
- Note: similar driver required for each device/transport
  - 1284, USB, 1394, 802.3, 802.11, Bluetooth, etc.



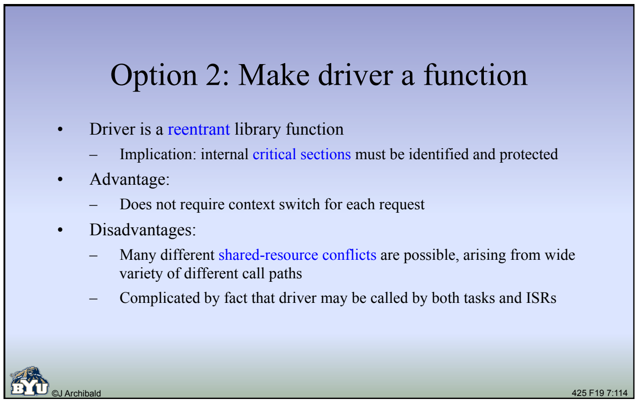
What form should driver take?

- ISR?
  - Not a viable option – can be ruled out
  - Driver actions too lengthy and complicated to be in interrupt code
  - ISR can't get semaphores, allocate memory, receive messages, etc.
- Task?
  - Explored as option 1 in following slides
- Library function called by clients (both ISRs and tasks)?
  - Explored as option 2



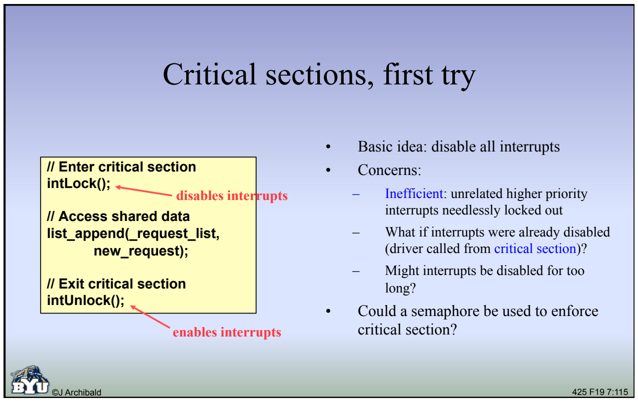
Option 1: Make driver a task

- Use a message queue to handle requests
  - Clients ask driver to take actions by sending it messages
  - Driver task simply empties message queue, services requests
- Advantage:
  - Relatively straightforward implementation
- Disadvantages:
  - Servicing every request causes a context switch
  - Serious problem if caller ever finds message queue full



Option 2: Make driver a function

- Driver is a reentrant library function
  - Implication: internal critical sections must be identified and protected
- Advantage:
  - Does not require context switch for each request
- Disadvantages:
  - Many different shared-resource conflicts are possible, arising from wide variety of different call paths
  - Complicated by fact that driver may be called by both tasks and ISRs

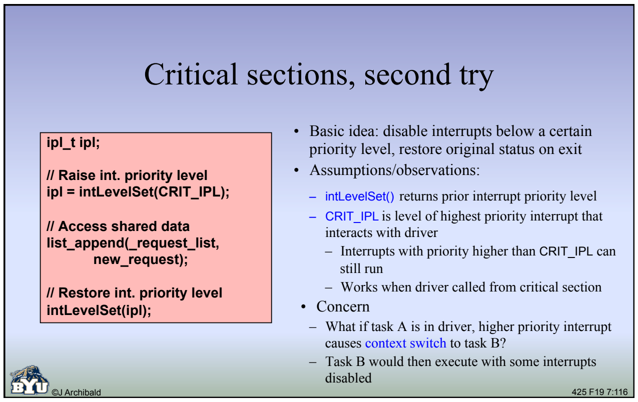


Critical sections, first try

```
// Enter critical section
intLock();

// Access shared data
list_append(_request_list,
        new_request);

// Exit critical section
intUnlock();
```

disables interrupts
enables interrupts

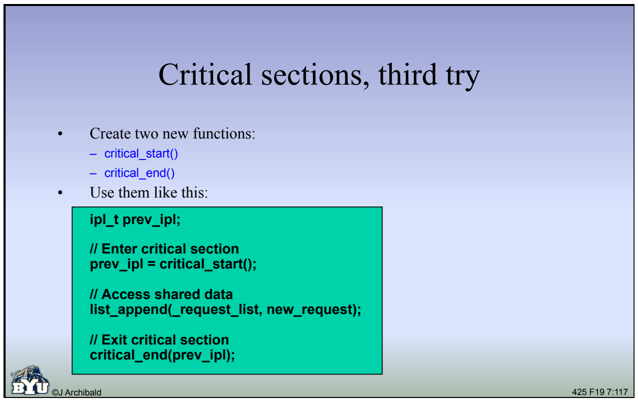
- Basic idea: disable all interrupts
- Concerns:
  - Inefficient: unrelated higher priority interrupts needlessly locked out
  - What if interrupts were already disabled (driver called from critical section)?
  - Might interrupts be disabled for too long?
- Could a semaphore be used to enforce critical section?



Critical sections, second try

```
ipl_t ipl;

// Raise int. priority level
ipl = intLevelSet(CRIT_IPL);

// Access shared data
list_append(_request_list,
        new_request);

// Restore int. priority level
intLevelSet(ipl);
```

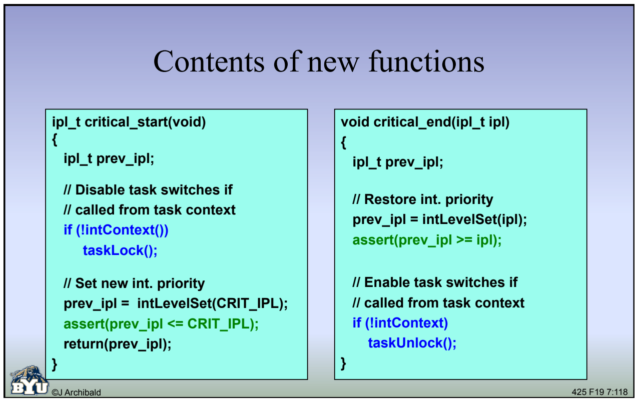
- Basic idea: disable interrupts below a certain priority level, restore original status on exit
- Assumptions/observations:
  - intLevelSet() returns prior interrupt priority level
  - CRIT_IPL is level of highest priority interrupt that interacts with driver
    - Interrupts with priority higher than CRIT_IPL can still run
    - Works when driver called from critical section
- Concern
  - What if task A is in driver, higher priority interrupt causes context switch to task B?
  - Task B would then execute with some interrupts disabled



Critical sections, third try

- Create two new functions:
  - critical_start()
  - critical_end()
- Use them like this:

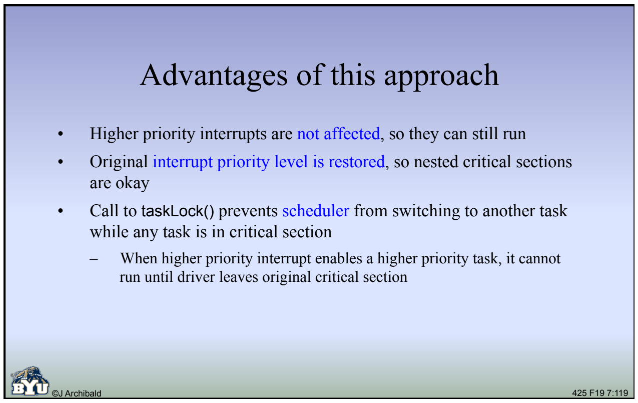
```
ipl_t prev_ipl;

// Enter critical section
prev_ipl = critical_start();

// Access shared data
list_append(_request_list, new_request);

// Exit critical section
critical_end(prev_ipl);
```



Contents of new functions

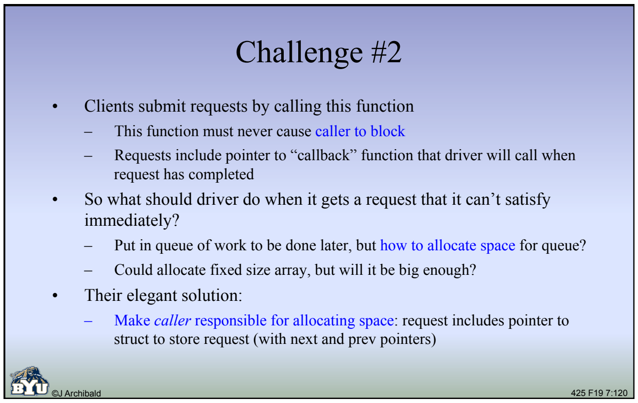
```
ipl_t critical_start(void)
{
  ipl_t prev_ipl;

  // Disable task switches if
  // called from task context
  if (!intContext())
     taskLock();

  // Set new int. priority
  prev_ipl =  intLevelSet(CRIT_IPL);
  assert(prev_ipl <= CRIT_IPL);
  return(prev_ipl);
}
```

```
void critical_end(ipl_t ipl)
{
  ipl_t prev_ipl;

  // Restore int. priority
  prev_ipl = intLevelSet(ipl);
  assert(prev_ipl >= ipl);

  // Enable task switches if
  // called from task context
  if (!intContext)
     taskUnlock();
}
```



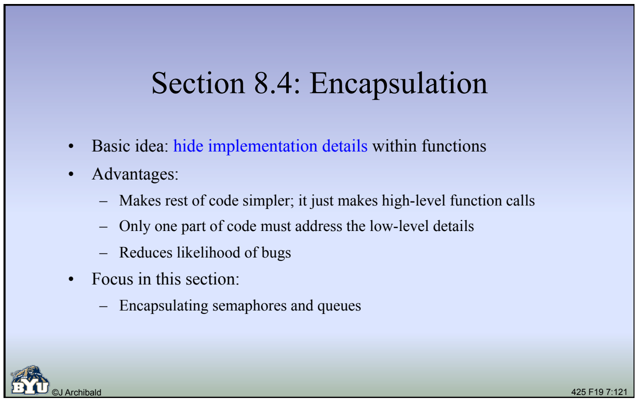
Advantages of this approach

- Higher priority interrupts are not affected, so they can still run
- Original interrupt priority level is restored, so nested critical sections are okay
- Call to taskLock() prevents scheduler from switching to another task while any task is in critical section
  - When higher priority interrupt enables a higher priority task, it cannot run until driver leaves original critical section



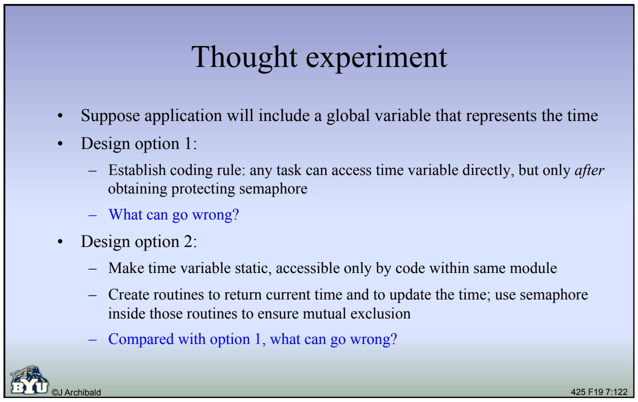
Challenge #2

- Clients submit requests by calling this function
  - This function must never cause caller to block
  - Requests include pointer to “callback” function that driver will call when request has completed
- So what should driver do when it gets a request that it can’t satisfy immediately?
  - Put in queue of work to be done later, but how to allocate space for queue?
  - Could allocate fixed size array, but will it be big enough?
- Their elegant solution:
  - Make *caller* responsible for allocating space: request includes pointer to struct to store request (with next and prev pointers)



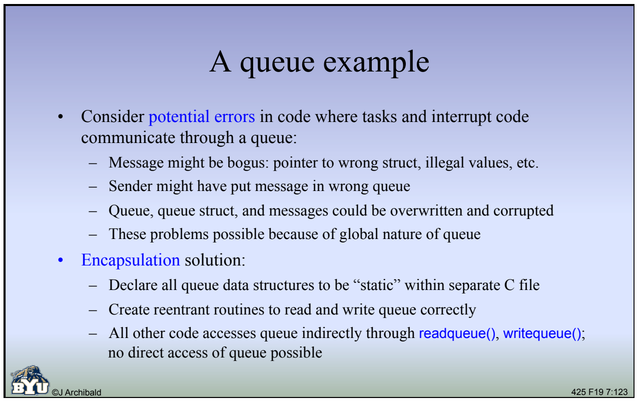
Section 8.4: Encapsulation

- Basic idea: hide implementation details within functions
- Advantages:
  - Makes rest of code simpler; it just makes high-level function calls
  - Only one part of code must address the low-level details
  - Reduces likelihood of bugs
- Focus in this section:
  - Encapsulating semaphores and queues



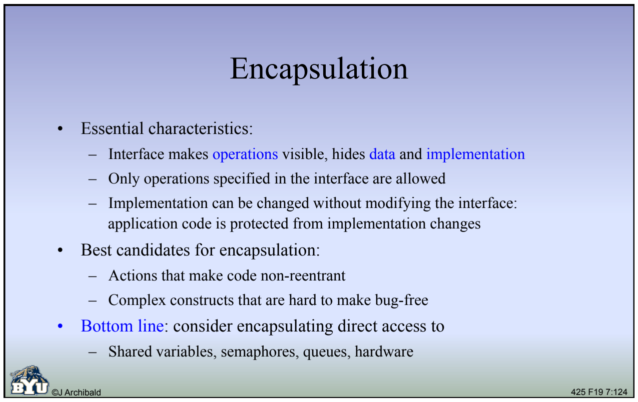
Thought experiment

- Suppose application will include a global variable that represents the time
- Design option 1:
  - Establish coding rule: any task can access time variable directly, but only *after* obtaining protecting semaphore
  - What can go wrong?
- Design option 2:
  - Make time variable static, accessible only by code within same module
  - Create routines to return current time and to update the time; use semaphore inside those routines to ensure mutual exclusion
  - Compared with option 1, what can go wrong?



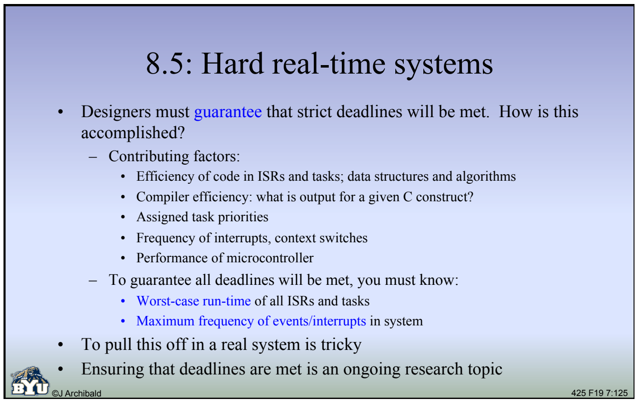
A queue example

- Consider potential errors in code where tasks and interrupt code communicate through a queue:
  - Message might be bogus: pointer to wrong struct, illegal values, etc.
  - Sender might have put message in wrong queue
  - Queue, queue struct, and messages could be overwritten and corrupted
  - These problems possible because of global nature of queue
- Encapsulation solution:
  - Declare all queue data structures to be "static" within separate C file
  - Create reentrant routines to read and write queue correctly
  - All other code accesses queue indirectly through readqueue(), writequeue(); no direct access of queue possible



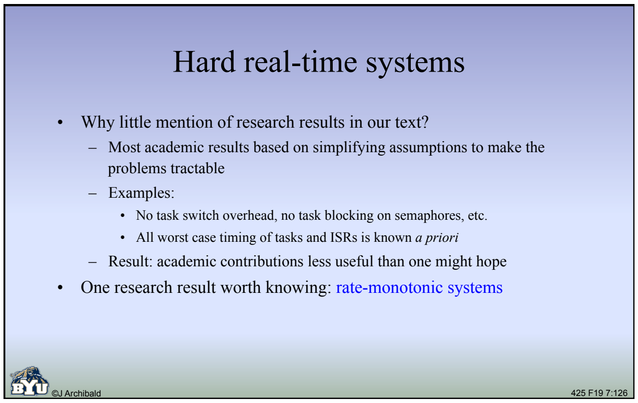
Encapsulation

- Essential characteristics:
  - Interface makes operations visible, hides data and implementation
  - Only operations specified in the interface are allowed
  - Implementation can be changed without modifying the interface: application code is protected from implementation changes
- Best candidates for encapsulation:
  - Actions that make code non-reentrant
  - Complex constructs that are hard to make bug-free
- Bottom line: consider encapsulating direct access to
  - Shared variables, semaphores, queues, hardware



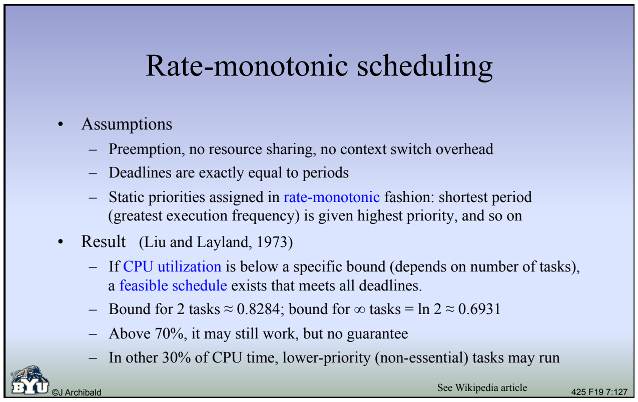
8.5: Hard real-time systems

- Designers must guarantee that strict deadlines will be met. How is this accomplished?
  - Contributing factors:
    - Efficiency of code in ISRs and tasks; data structures and algorithms
    - Compiler efficiency: what is output for a given C construct?
    - Assigned task priorities
    - Frequency of interrupts, context switches
    - Performance of microcontroller
  - To guarantee all deadlines will be met, you must know:
    - Worst-case run-time of all ISRs and tasks
    - Maximum frequency of events/interrupts in system
- To pull this off in a real system is tricky
- Ensuring that deadlines are met is an ongoing research topic



Hard real-time systems

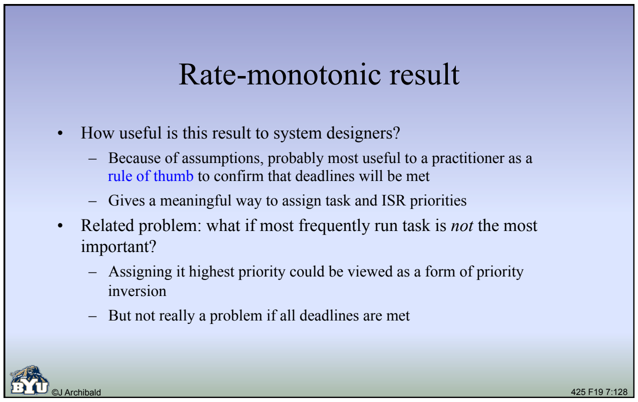
- Why little mention of research results in our text?
  - Most academic results based on simplifying assumptions to make the problems tractable
  - Examples:
    - No task switch overhead, no task blocking on semaphores, etc.
    - All worst case timing of tasks and ISRs is known *a priori*
  - Result: academic contributions less useful than one might hope
- One research result worth knowing: rate-monotonic systems



Rate-monotonic scheduling

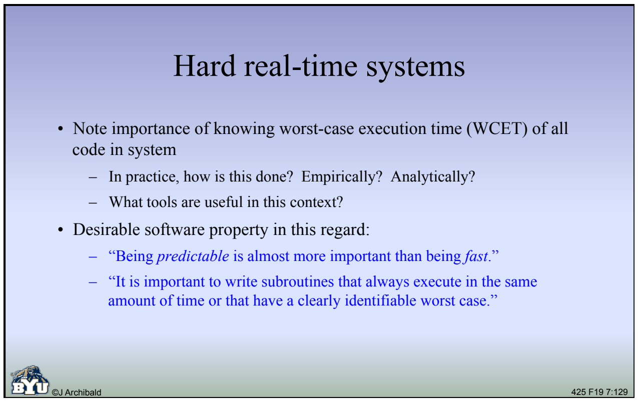
- Assumptions
  - Preemption, no resource sharing, no context switch overhead
  - Deadlines are exactly equal to periods
  - Static priorities assigned in rate-monotonic fashion: shortest period (greatest execution frequency) is given highest priority, and so on
- Result (Liu and Layland, 1973)
  - If CPU utilization is below a specific bound (depends on number of tasks), a feasible schedule exists that meets all deadlines.
  - Bound for 2 tasks ≈ 0.8284; bound for ∞ tasks = ln 2 ≈ 0.6931
  - Above 70%, it may still work, but no guarantee
  - In other 30% of CPU time, lower-priority (non-essential) tasks may run

See Wikipedia article



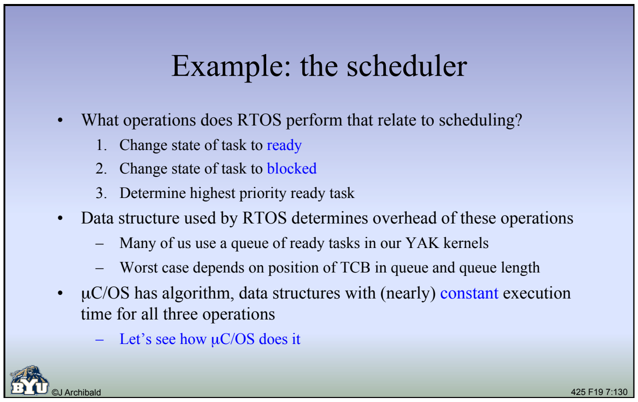
Rate-monotonic result

- How useful is this result to system designers?
  - Because of assumptions, probably most useful to a practitioner as a rule of thumb to confirm that deadlines will be met
  - Gives a meaningful way to assign task and ISR priorities
- Related problem: what if most frequently run task is *not* the most important?
  - Assigning it highest priority could be viewed as a form of priority inversion
  - But not really a problem if all deadlines are met



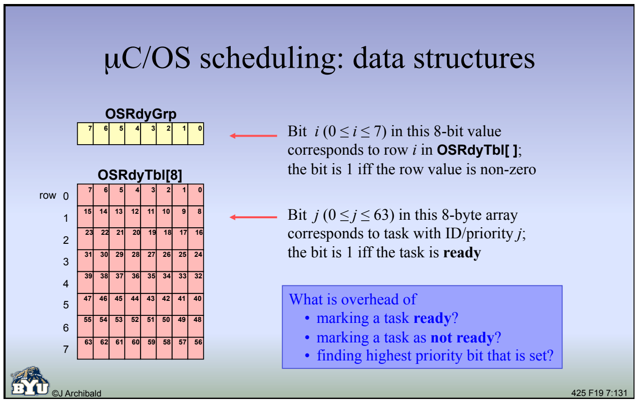
Hard real-time systems

- Note importance of knowing worst-case execution time (WCET) of all code in system
  - In practice, how is this done? Empirically? Analytically?
  - What tools are useful in this context?
- Desirable software property in this regard:
  - "Being *predictable* is almost more important than being *fast*."
  - "It is important to write subroutines that always execute in the same amount of time or that have a clearly identifiable worst case."



Example: the scheduler

- What operations does RTOS perform that relate to scheduling?
  1. Change state of task to ready
  2. Change state of task to blocked
  3. Determine highest priority ready task
- Data structure used by RTOS determines overhead of these operations
  - Many of us use a queue of ready tasks in our YAK kernels
  - Worst case depends on position of TCB in queue and queue length
- µC/OS has algorithm, data structures with (nearly) constant execution time for all three operations
  - Let's see how µC/OS does it



µC/OS scheduling: data structures

OSRdyGrp

7	6	5	4	3	2	1	0

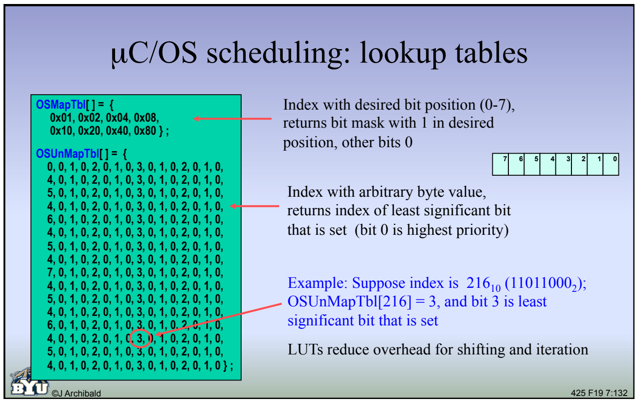
Bit *i* ($0 \le i \le 7$) in this 8-bit value corresponds to row *i* in **OSRdyTbl[ ]**; the bit is 1 iff the row value is non-zero

OSRdyTbl[8]

row								
0	7	6	5	4	3	2	1	0
1	15	14	13	12	11	10	9	8
2	23	22	21	20	19	18	17	16
3	31	30	29	28	27	26	25	24
4	39	38	37	36	35	34	33	32
5	47	46	45	44	43	42	41	40
6	55	54	53	52	51	50	49	48
7	63	62	61	60	59	58	57	56

Bit *j* ($0 \le j \le 63$) in this 8-byte array corresponds to task with ID/priority *j*; the bit is 1 iff the task is **ready**

What is overhead of
- marking a task **ready**?
- marking a task as **not ready**?
- finding highest priority bit that is set?



µC/OS scheduling: lookup tables

```
OSMapTbl[ ] = {
    0x01, 0x02, 0x04, 0x08,
    0x10, 0x20, 0x40, 0x80 } ;

OSUnMapTbl[ ] = {
   0, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
   4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
   5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
   4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
   6, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
   4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
   5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
   4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
   7, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
   4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
   5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
   4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
   6, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
   4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
   5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
   4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0 } ;
```

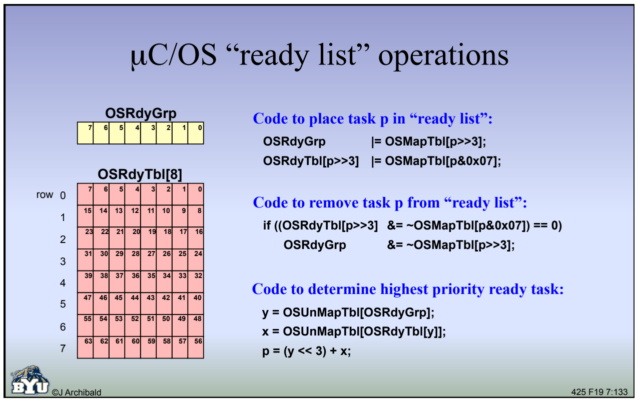
Index with desired bit position (0-7), returns bit mask with 1 in desired position, other bits 0

7	6	5	4	3	2	1	0

Index with arbitrary byte value, returns index of least significant bit that is set (bit 0 is highest priority)

Example: Suppose index is 216_{10} (11011000_2); OSUnMapTbl[216] = 3, and bit 3 is least significant bit that is set

LUTs reduce overhead for shifting and iteration



μC/OS “ready list” operations

OSRdyGrp

7	6	5	4	3	2	1	0

OSRdyTbl[8]

row								
0	7	6	5	4	3	2	1	0
1	15	14	13	12	11	10	9	8
2	23	22	21	20	19	18	17	16
3	31	30	29	28	27	26	25	24
4	39	38	37	36	35	34	33	32
5	47	46	45	44	43	42	41	40
6	55	54	53	52	51	50	49	48
7	63	62	61	60	59	58	57	56

Code to place task p in “ready list”:

```
OSRdyGrp        |= OSMapTbl[p>>3];
OSRdyTbl[p>>3]  |= OSMapTbl[p&0x07];
```

Code to remove task p from “ready list”:

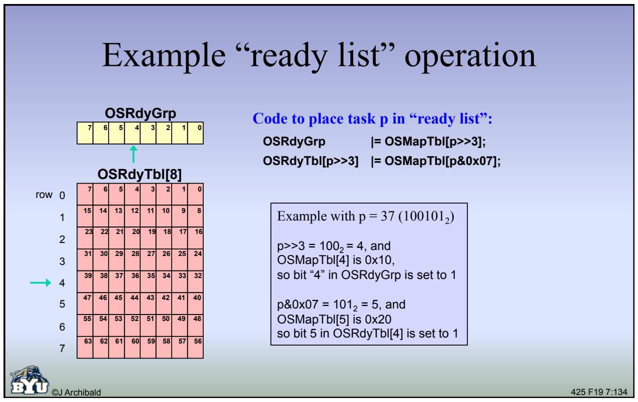
```
if ((OSRdyTbl[p>>3]  &= ~OSMapTbl[p&0x07]) == 0)
    OSRdyGrp         &= ~OSMapTbl[p>>3];
```

Code to determine highest priority ready task:

```
y = OSUnMapTbl[OSRdyGrp];
x = OSUnMapTbl[OSRdyTbl[y]];
p = (y << 3) + x;
```

©J Archibald

425 F19 7:133



Example “ready list” operation

OSRdyGrp

7	6	5	4	3	2	1	0

OSRdyTbl[8]

row								
0	7	6	5	4	3	2	1	0
1	15	14	13	12	11	10	9	8
2	23	22	21	20	19	18	17	16
3	31	30	29	28	27	26	25	24
→ 4	39	38	37	36	35	34	33	32
5	47	46	45	44	43	42	41	40
6	55	54	53	52	51	50	49	48
7	63	62	61	60	59	58	57	56

Code to place task p in “ready list”:

```
OSRdyGrp        |= OSMapTbl[p>>3];
OSRdyTbl[p>>3]  |= OSMapTbl[p&0x07];
```

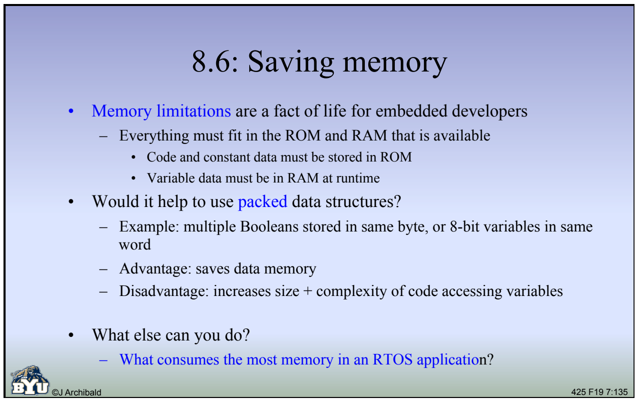
Example with p = 37 (100101_2)

p>>3 = 100_2 = 4, and
OSMapTbl[4] is 0x10,
so bit “4” in OSRdyGrp is set to 1

p&0x07 = 101_2 = 5, and
OSMapTbl[5] is 0x20
so bit 5 in OSRdyTbl[4] is set to 1

©J Archibald

425 F19 7:134

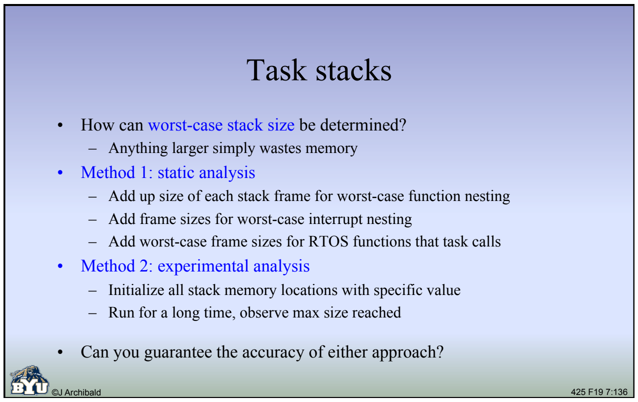


8.6: Saving memory

- Memory limitations are a fact of life for embedded developers
  - Everything must fit in the ROM and RAM that is available
    - Code and constant data must be stored in ROM
    - Variable data must be in RAM at runtime
- Would it help to use packed data structures?
  - Example: multiple Booleans stored in same byte, or 8-bit variables in same word
  - Advantage: saves data memory
  - Disadvantage: increases size + complexity of code accessing variables
- What else can you do?
  - What consumes the most memory in an RTOS application?

©J Archibald

425 F19 7:135

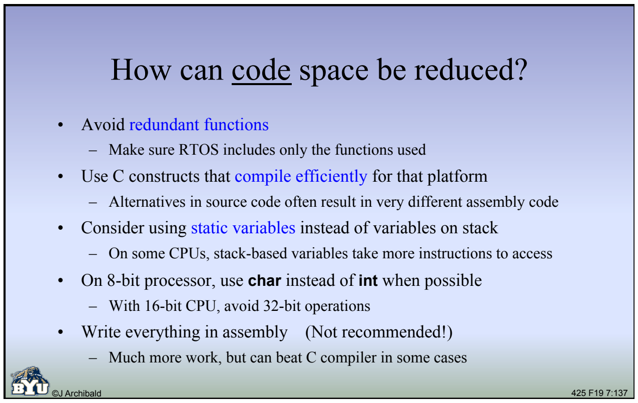


Task stacks

- How can worst-case stack size be determined?
  - Anything larger simply wastes memory
- Method 1: static analysis
  - Add up size of each stack frame for worst-case function nesting
  - Add frame sizes for worst-case interrupt nesting
  - Add worst-case frame sizes for RTOS functions that task calls
- Method 2: experimental analysis
  - Initialize all stack memory locations with specific value
  - Run for a long time, observe max size reached
- Can you guarantee the accuracy of either approach?

©J Archibald

425 F19 7:136

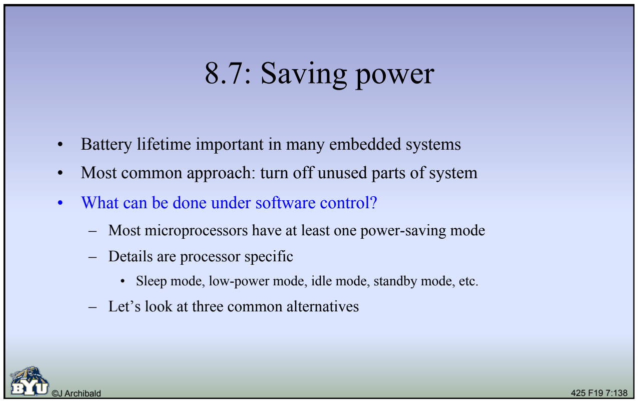


How can code space be reduced?

- Avoid redundant functions
  - Make sure RTOS includes only the functions used
- Use C constructs that compile efficiently for that platform
  - Alternatives in source code often result in very different assembly code
- Consider using static variables instead of variables on stack
  - On some CPUs, stack-based variables take more instructions to access
- On 8-bit processor, use **char** instead of **int** when possible
  - With 16-bit CPU, avoid 32-bit operations
- Write everything in assembly (Not recommended!)
  - Much more work, but can beat C compiler in some cases

©J Archibald

425 F19 7:137

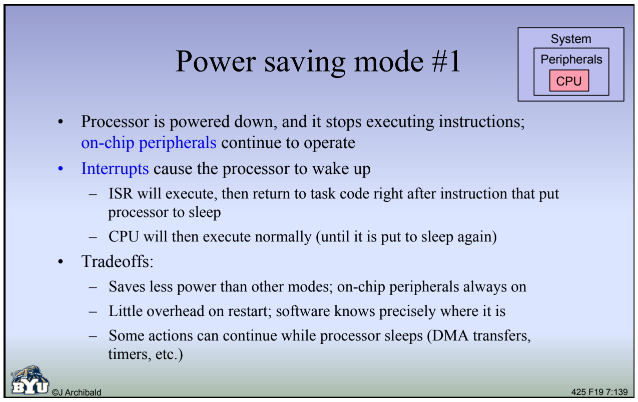


8.7: Saving power

- Battery lifetime important in many embedded systems
- Most common approach: turn off unused parts of system
- What can be done under software control?
  - Most microprocessors have at least one power-saving mode
  - Details are processor specific
    - Sleep mode, low-power mode, idle mode, standby mode, etc.
  - Let’s look at three common alternatives

©J Archibald

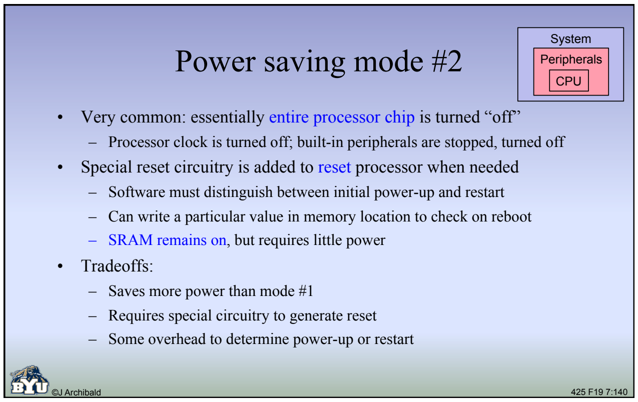
425 F19 7:138



Power saving mode #1

System
Peripherals
CPU

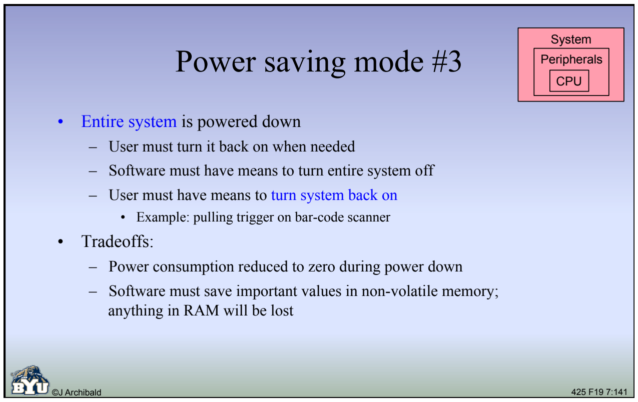
- Processor is powered down, and it stops executing instructions; on-chip peripherals continue to operate
- Interrupts cause the processor to wake up
  - ISR will execute, then return to task code right after instruction that put processor to sleep
  - CPU will then execute normally (until it is put to sleep again)
- Tradeoffs:
  - Saves less power than other modes; on-chip peripherals always on
  - Little overhead on restart; software knows precisely where it is
  - Some actions can continue while processor sleeps (DMA transfers, timers, etc.)



Power saving mode #2

System
Peripherals
CPU

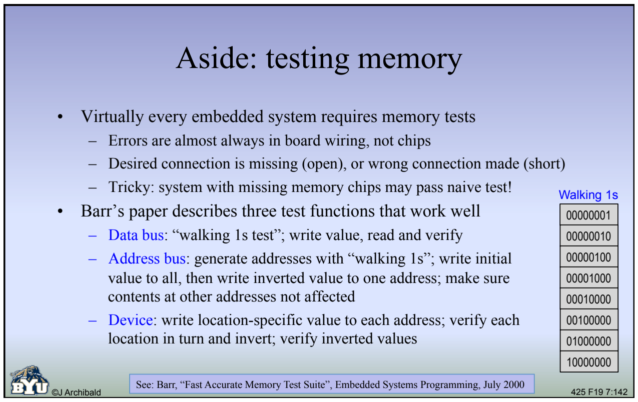
- Very common: essentially entire processor chip is turned "off"
  - Processor clock is turned off; built-in peripherals are stopped, turned off
- Special reset circuitry is added to reset processor when needed
  - Software must distinguish between initial power-up and restart
  - Can write a particular value in memory location to check on reboot
  - SRAM remains on, but requires little power
- Tradeoffs:
  - Saves more power than mode #1
  - Requires special circuitry to generate reset
  - Some overhead to determine power-up or restart



Power saving mode #3

System
Peripherals
CPU

- Entire system is powered down
  - User must turn it back on when needed
  - Software must have means to turn entire system off
  - User must have means to turn system back on
    - Example: pulling trigger on bar-code scanner
- Tradeoffs:
  - Power consumption reduced to zero during power down
  - Software must save important values in non-volatile memory; anything in RAM will be lost

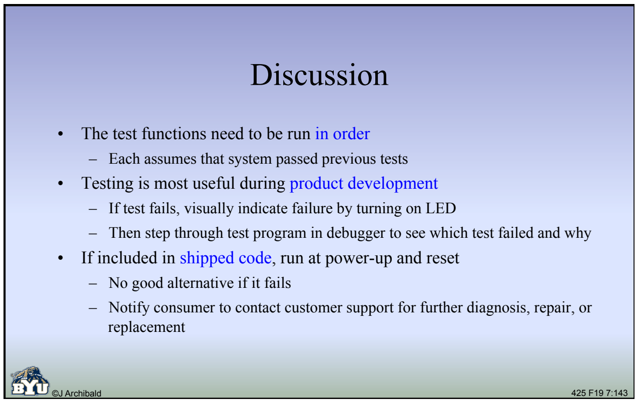


Aside: testing memory

- Virtually every embedded system requires memory tests
  - Errors are almost always in board wiring, not chips
  - Desired connection is missing (open), or wrong connection made (short)
  - Tricky: system with missing memory chips may pass naive test!
- Barr's paper describes three test functions that work well
  - Data bus: "walking 1s test"; write value, read and verify
  - Address bus: generate addresses with "walking 1s"; write initial value to all, then write inverted value to one address; make sure contents at other addresses not affected
  - Device: write location-specific value to each address; verify each location in turn and invert; verify inverted values

Walking 1s
00000001
00000010
00000100
00001000
00010000
00100000
01000000
10000000

See: Barr, "Fast Accurate Memory Test Suite", Embedded Systems Programming, July 2000



Discussion

- The test functions need to be run in order
  - Each assumes that system passed previous tests
- Testing is most useful during product development
  - If test fails, visually indicate failure by turning on LED
  - Then step through test program in debugger to see which test failed and why
- If included in shipped code, run at power-up and reset
  - No good alternative if it fails
  - Notify consumer to contact customer support for further diagnosis, repair, or replacement