Teaching Assessment (OMET)

- Your feedback is important
  - What do you like about this course?
  - Suggestions for improvement?

- Incentive
  - 1% of your **<u>final</u>** course grade
  - Send me the screenshot of completion

Recap from last class

- Dynamic power management policy
  - Workload-independent metrics for energy saving calculation
  - Predictive techniques
    - Fixed timeout policy
    - Predictive shutdown and wakeup

- Power manager
  - Advanced Configuration and Power Interface (ACPI)

- Holistic approach
  - Memory system
  - Cache behavior
  - Optimizing operations on I/O devices

ECE 1175
Embedded System Design

Safety and Reliability

## Wei Gao

What is a "Safe" Embedded System?

- Again, the ABS system's case

- Is it safe?

- What does "safe" mean?
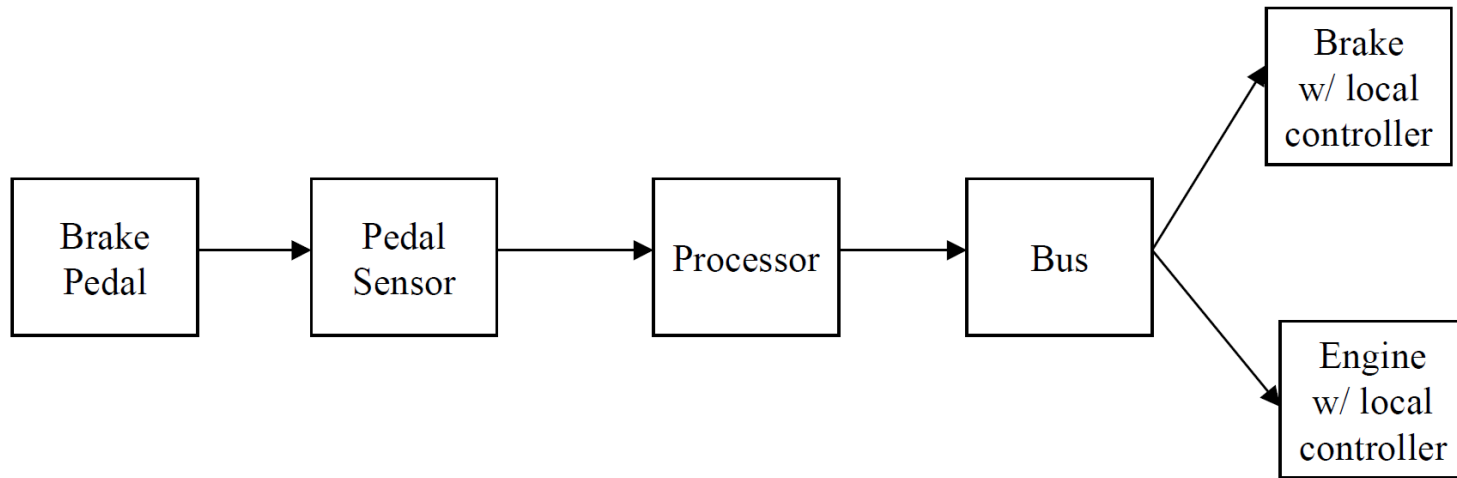
- How can we make it safe?

- Add watch dog between brake and bus

- Add mechanical linkage from brake pedal directly to brake
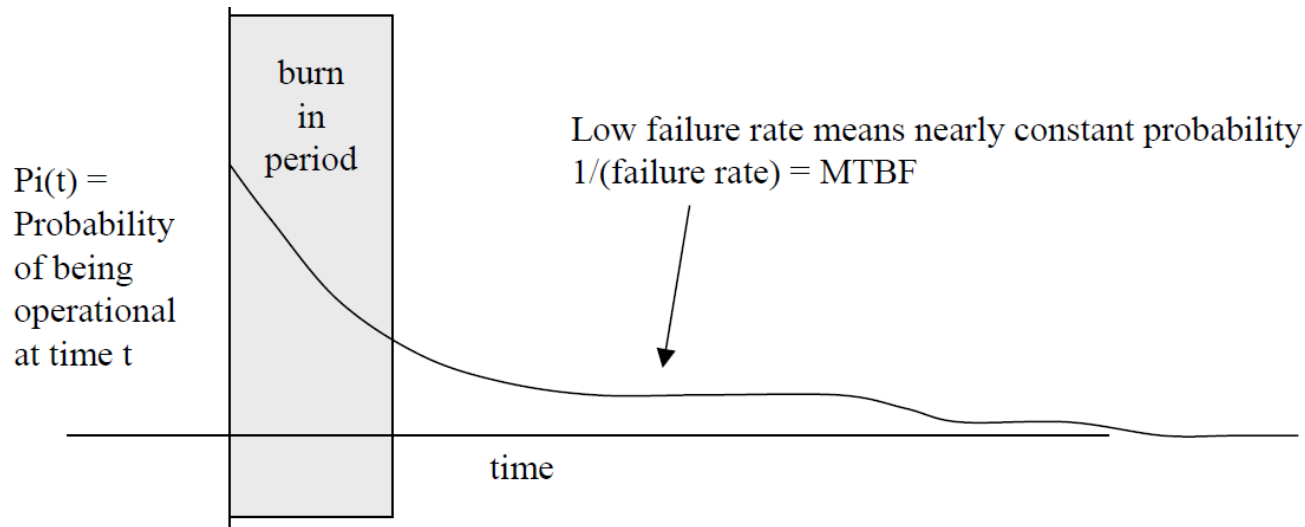
- Add a third mechanical linkage…

A "Safe" Embedded System

- A system is **safe** if its deployment involves assuming an *acceptable* amount of risk…acceptable to whom?

- Risk factors
    - Probability of something bad happing
    - Consequences of something bad happening (Severity)

- Example
    - Airplane Travel – high severity, low probability
    - Electric shock from battery powered devices – high probability, low severity

5
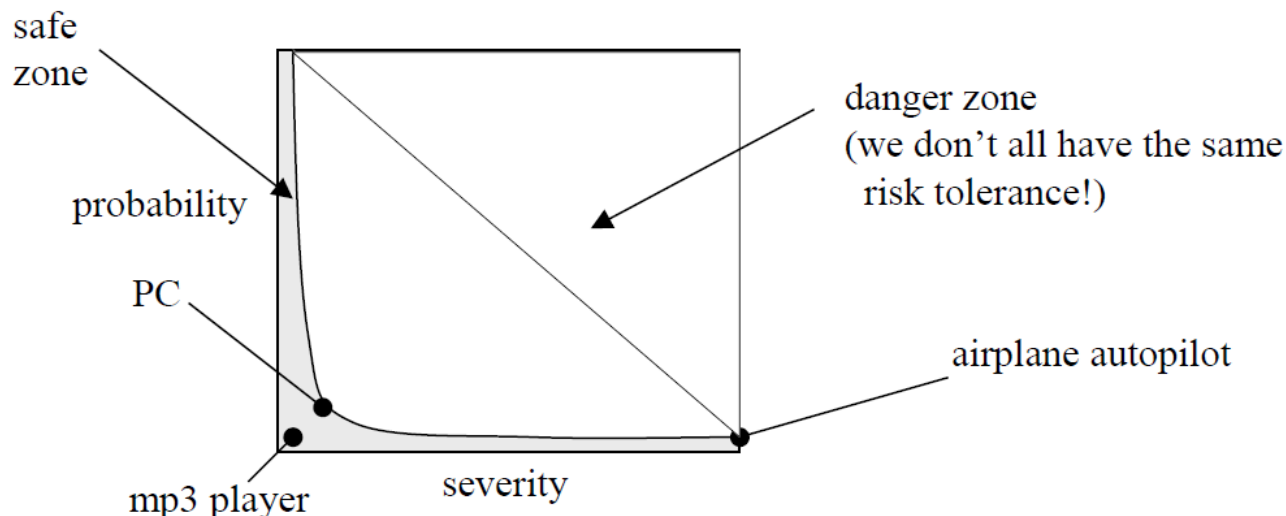
A "Reliable" Embedded System

- **Reliability** of component *i* can be expressed as the probability that component *i* is still functioning at some time *t*

$Pi(t) =$ Probability of being operational at time t

burn in period

Low failure rate means nearly constant probability
1/(failure rate) = MTBF

time

- Is the system's reliability $P_s(t) = \prod_i P_i(t)$?
    - Assuming that all components have the same component reliability, Is a system w/ fewer components always more reliable?
    - Does component failure always result in system failure?

Faults

- No problem if there are no faults!
- A **fault** is an "unsatisfactory system condition or state"
- Systematic faults
  - Design Errors (includes process errors such as failure to test or failure to apply a safety design process)
  - Faults due to software bugs are systemic
  - Security breech
- Random faults
  - Random events that can cause permanent or temporary damage to the system. Includes EMI and radiation, component failure, power supply problems, wear and tear.

Component vs. System

- Reliability is a component issue
- **Safety** and **Availability** are system issues
- A system can be safe even if it is unreliable!

- If a system has lots of redundancy the likelihood of a component failure (a fault) increases, but so may increase the safety and availability of that system

- Safety and Availability are different and sometimes at odds. Safety may require the shutdown of a system that may still be able to perform its function.
  - A backup system that can fully operate a nuclear power plant might always shut it down in the event of failure of the primary system.
  - The plant could remain available, but it is unsafe to continue operation

Terms

- Safety: assuming acceptable risk

- Hazard: dangerous system state that could cause damage

- Fault: conditions that lead to hazards

- Reliability

  - System is functioning if all components are functioning
    - $P_s(t) = \prod_i P_i(t)$

  - System is functioning
    - $P_s(t) = 1 - \prod_i F_i(t)$
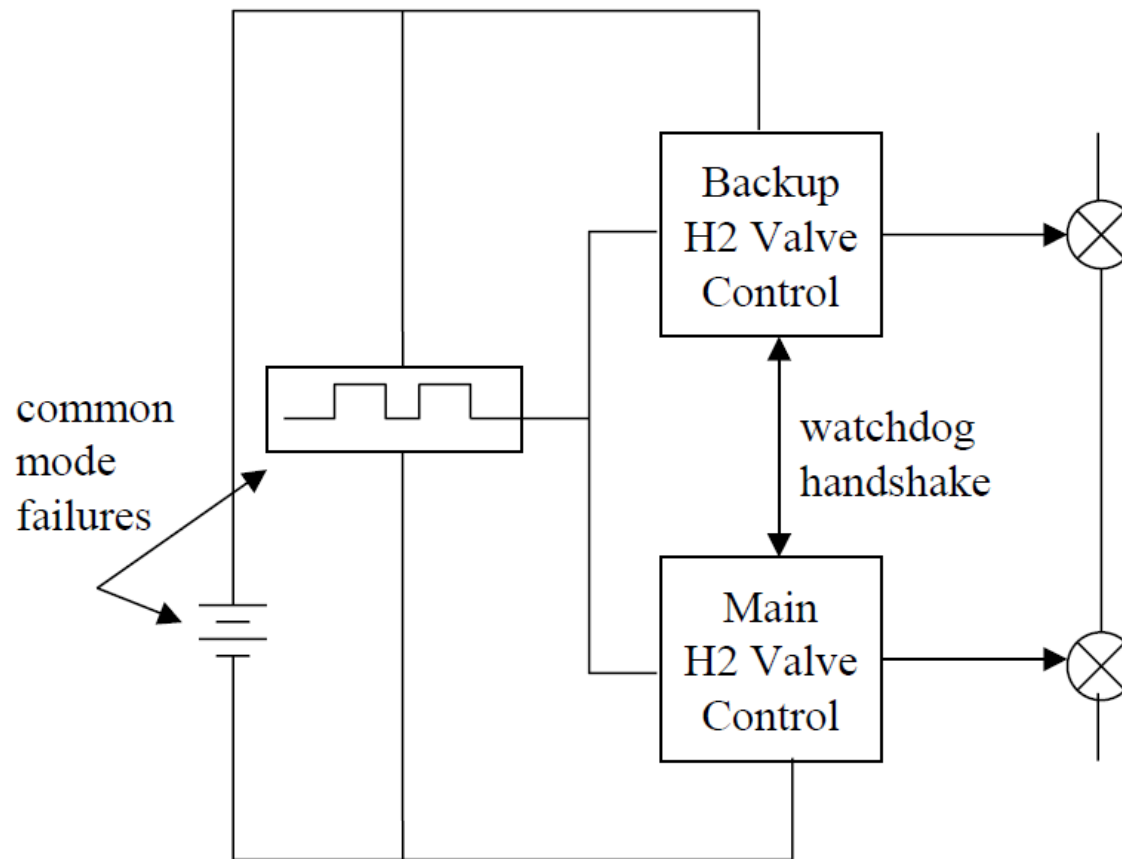   - $F_{i(t)} = 1 - Pi(t)$: probability of component failure

Single Fault Tolerance (for Safety)

- Single fault tolerant systems are generally considered to be safe, but more stringent requirements may apply to high risk cases…airplanes, power plants, etc.
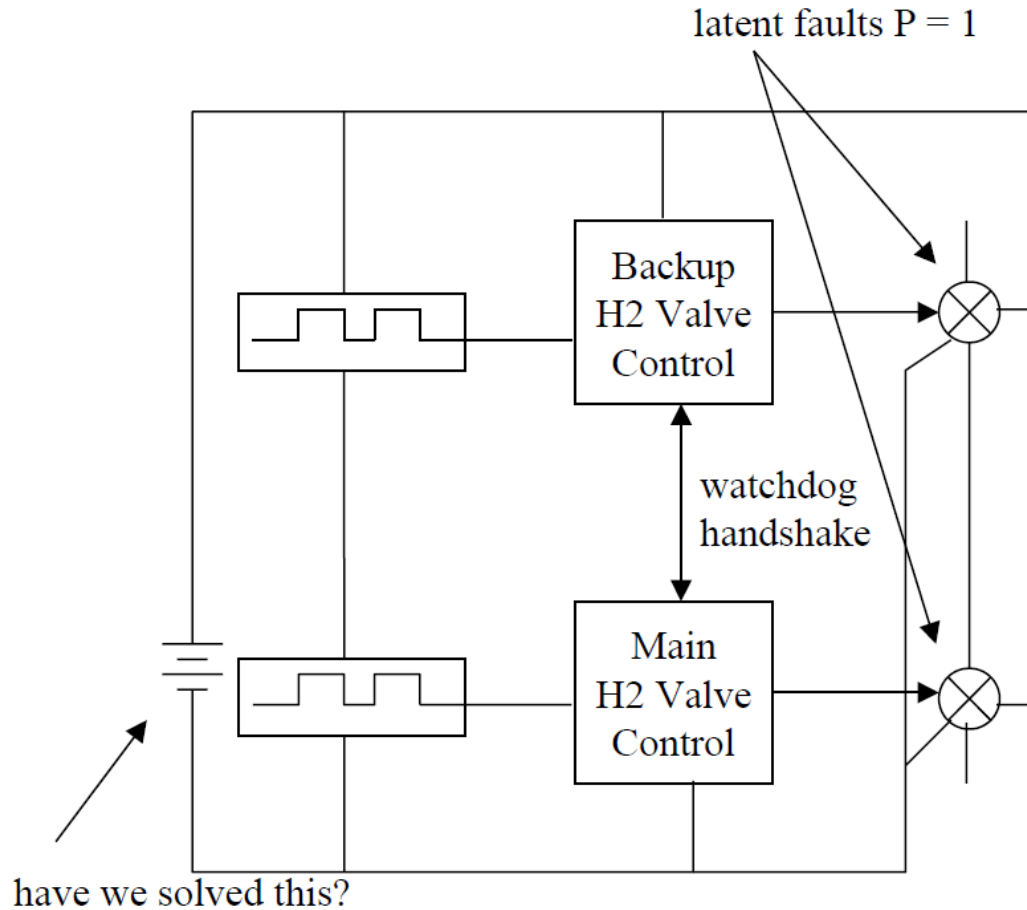


```
Backup H2 Valve Control

watchdog protocol

Main H2 Valve Control
```
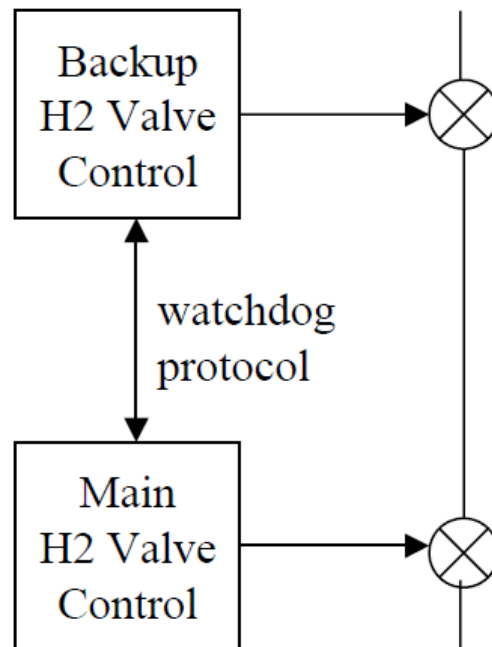
If the handshake fails, then either one or both can shut off the gas supply. Is this a single fault tolerant system?

Is This Safe?

Now Safe?

latent faults P = 1

Backup
H2 Valve
Control

watchdog
handshake

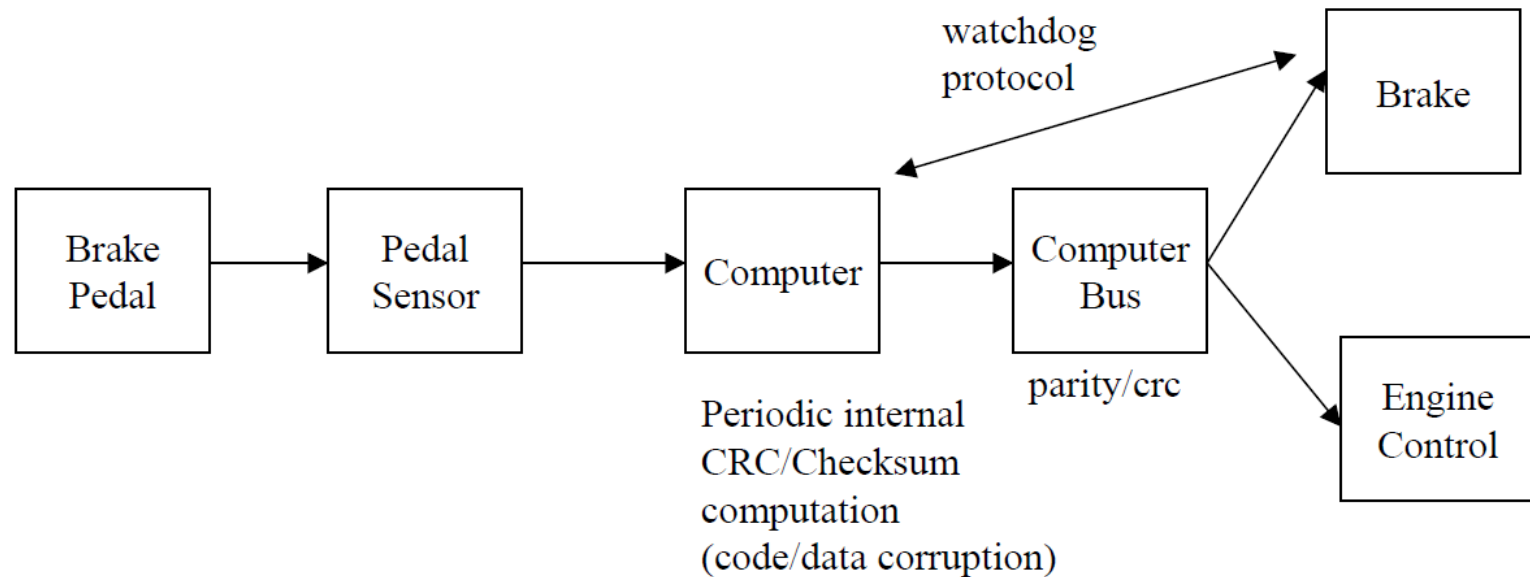Main
H2 Valve
Control

have we solved this?

- Separate Clock Source
- Power Fail-Safe (non-latching) Valves

What about power spike that confuses both processors at the same time? Maybe the watchdog can't be software based.

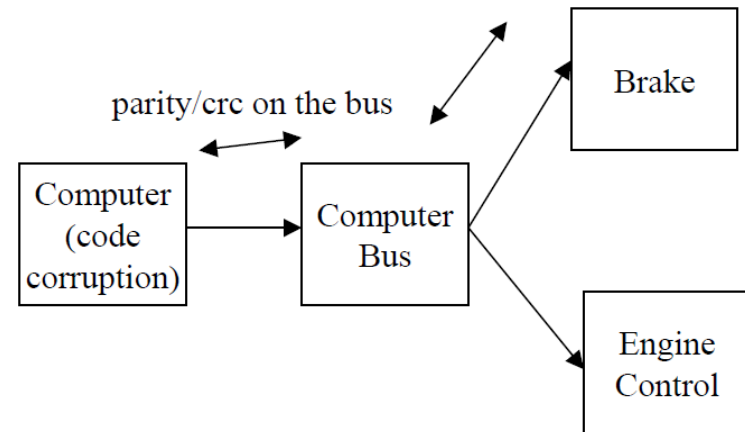Does it ever end?

Safety Architectures

- Self checking (single channel protected design)
- Redundancy
- Diversity or Heterogeneity

Single Channel Protection

- Self Checking
  - Perform periodic checksums on code and data
  - How long does it take?
  - No protection against systematic faults
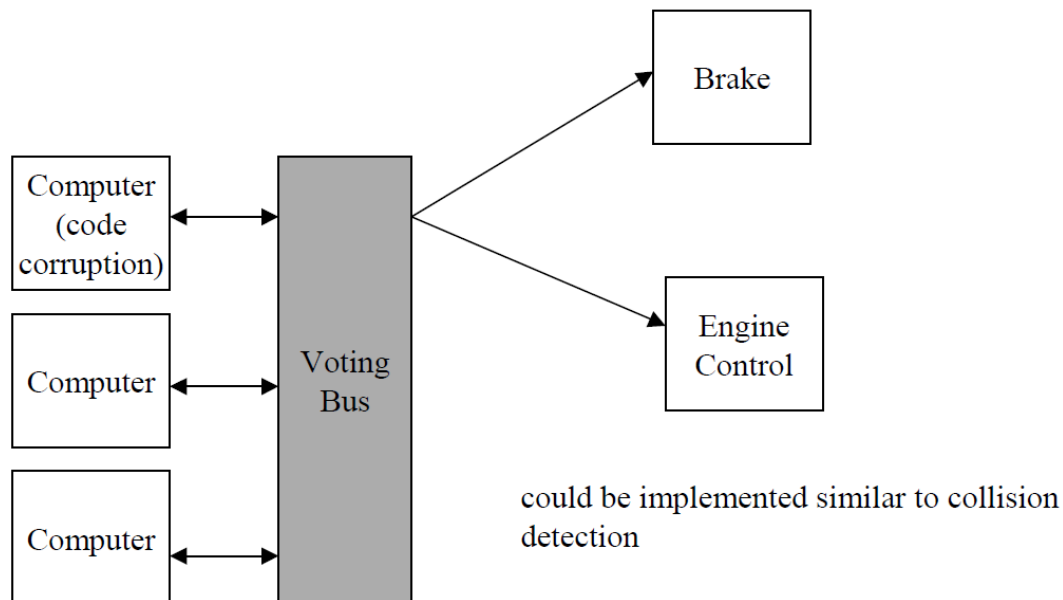
- Feasibility of single channel protection
  - Fault tolerance time
  - Processor speed
  - Memory size
  - Special hardware

parity/crc on the bus

Computer (code corruption) → Computer Bus → Brake

Computer Bus → Engine Control

# Redundancy

- Homogeneous redundancy
  - Low development cost – just duplicate
  - High recurring cost
  - No protection against systematic faults

Computer (code corruption) ⟷ Voting Bus → Brake

Computer ⟷ Voting Bus

Computer ⟷ Voting Bus → Engine Control

could be implemented similar to collision detection

Diversity

- Heterogeneous redundancy
  - Protects against random and some systematic faults
- Example: space shuttle – 5 computers, 4 same 1 different
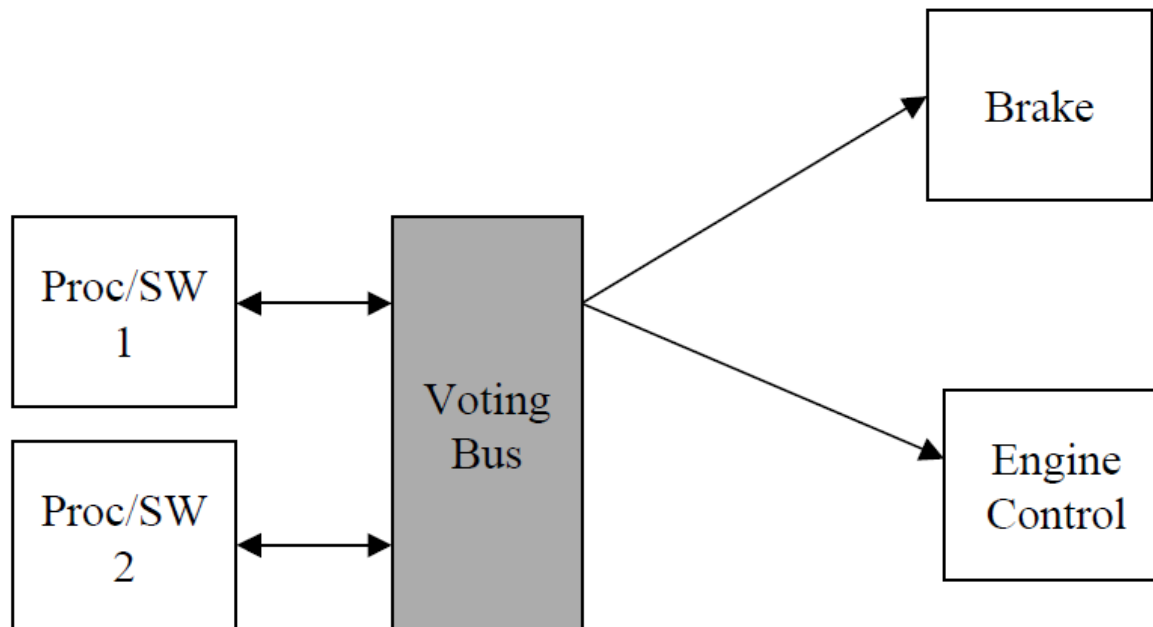
Design Process

1. Hazard identification and fault analysis

2. Risk assessment

3. Define safety measures

4. Create safety requirements

5. Implement safety

6. Test, Test, Test, Test, …

Language Definition

- Static analysis
  - Software: compilation time
  - Hardware: design phase

- Dynamic analysis at run-time
  - Make sure that you never access memory in the wrong way
    - Null pointer access
    - Array out of bounds
    - Type mismatch even when casting
    - Memory management and garbage collection
  - What happens in the event of an exception?

Exception Handling

- It is NOT ok to just let the system crash if some operation fails!
  - You must, at least, get into safe mode
- Language-dependent handling
- Typical C approach:
  - Exception flow is the same as normal flow
  - Use negative numbers for exceptions

```
a = f1(b,c)
if (a) switch (a) {
case 1: handle exception 1
case 2: handle exception 2
…
}
b = f2(e,f)
if (a) switch (a) {
case 1: handle exception 1
case 2: handle exception 2
…
}
```

Exception Handling in Java

- All exceptions must be handled or thrown

```
void myMethod() throws FatalException {
        try {
                a = x.f1(b,c)
                b = x.f2(e,f)
                if (a) … // handle all functional outcomes here!
        } catch (IOException e) {
                recover and continue if that's okay.
        } catch (ArrayOutOfBoundsException e) {
                not recoverable, throw new FatalException("I'm Dead");
        } finally {
                finish up and exit
        }
}
```

Encapsulation: semantic checking

- In C

```
while (item!=tail) {
            process(item);
            if (item->next == null) return –1 // exception ?
            item = item->next;
    }
```

- In Java

```
while (item = mylist.next()) { // inside mylist is not my problem
            process (item);
}
class list {
            Object next() throws CorruptListException {
            if (current == tail) return null;
                        current = current.next; // private field access okay
            if (current == null) throw new CorruptListException(this.toString());
            return(current.value);
}
```

Java for Embedded Systems

- Why not using Java for embedded systems?
  - It is slower
  - Code bloat
  - Garbage collection may not be interruptible
    - Long latency, low predictability
  - Low time resolution
  - Limited hardware access

Summary

- Safety and reliability
  - Component vs. system

- Techniques to ensure safety/reliability
  - Self checking
  - Redundancy
  - Diversity and heterogeneity

- Language definition
  - Be careful about exception handling and garbage collection