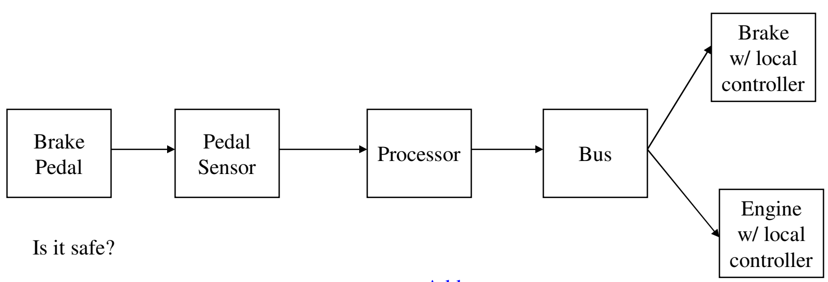
## Embedded Systems Safety

- Terms and Concepts
- Safety Architectures
- Safe Design Process
- Software Specific Stuff
- Sources
  - *Hard Time* by Bruce Powell Douglass, which references *Safeware* by Nancy Leveson
  - Adapted from Larry Arnstein



## What is a Safe System?




Is it safe?

Add electronic watch dog between brake and bus

What does "safe" mean?

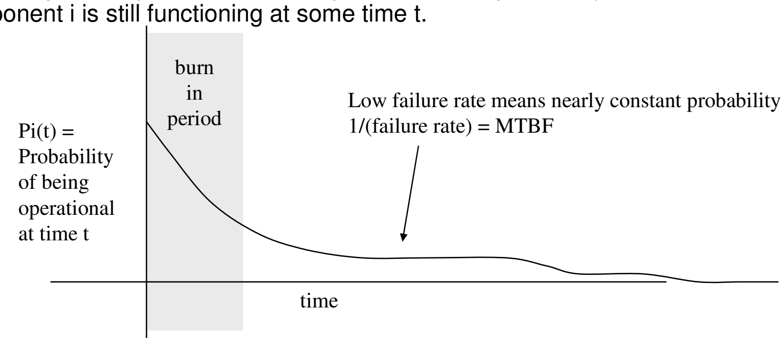
Add mechanical linkage from separate brake pedal directly to brake

Add a third mechanical linkage….

How can we make it safe?

## Terms and Concepts

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

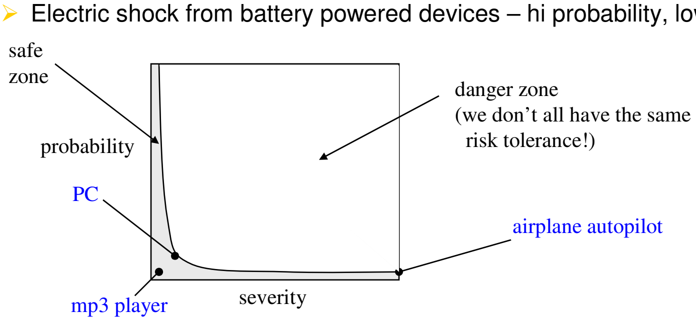



- Is system reliability $P_s(t) = \Pi P_i(t)$ ?
- Assuming that all components have the same component reliability, Is a system w/ fewer components always more reliable ?
- Does component failure → system failure ?



## A Safety System

- A system is **safe** if it's 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 – hi probability, low severity



## More Precise Terminology

- **Accident or Mishap**: (unintended) Damage to property or harm to persons. Economic impact of failure to meet warranted performance is outside of the scope of safety.
- **Hazard**: A state of the the system that will inevitably lead to an accident or mishap
  - Release of Energy
  - Release of Toxins
  - Interference with life support functions
  - Supplying misleading information to safety personnel or control systems. This is the desktop PC nightmare scenario. Bad information
  - Failure to alarm when hazardous conditions exist



## Faults

- **A fault** is an "unsatisfactory system condition or state". A fault is not necessarily a hazard. In fact, assessments of safety are based on the notion of *fault tolerance*.
- **Systemic** 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.

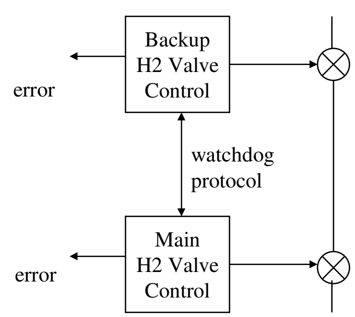
## Component v. System

- 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



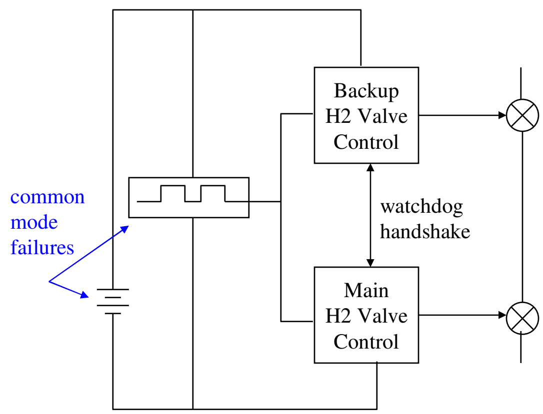
## Single Fault Tolerance (for safety)

- The existence of any single fault does not result in a hazard
- Single fault tolerant systems are generally considered to be safe, but more stringent requirements may apply to high risk cases…airplanes, power plants, etc.



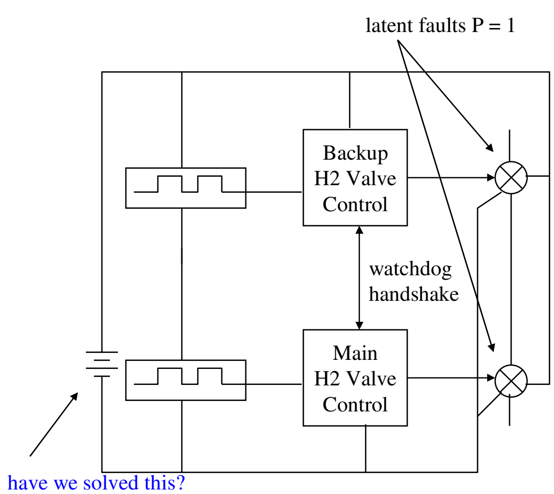

If the handshake fails, then either one or both can shut off the gas supply. Is this a single fault tolerant system?

## Is This?






## Now Safe?




- 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.

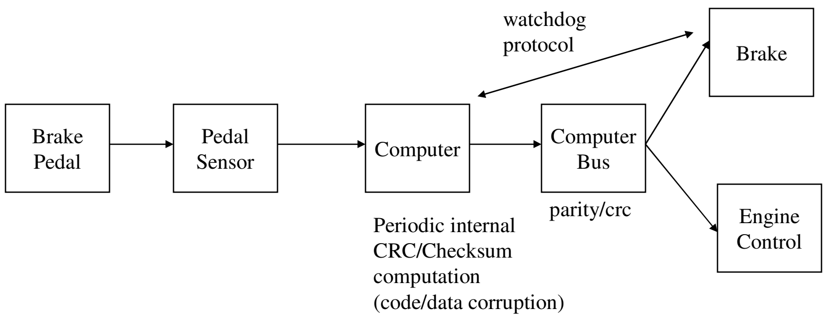
Does it ever end?

$$T_{test} < T_0 < T_1$$

detection time is < than single fault tolerance time < time to second failure

## Safety Architectures

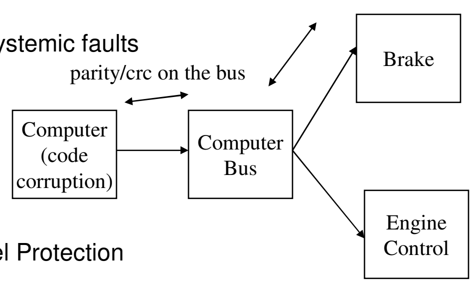
- Self Checking (Single Channel Protected Design)
- Redundancy
- Diversity or Heterogeneity






## Single Channel Protection

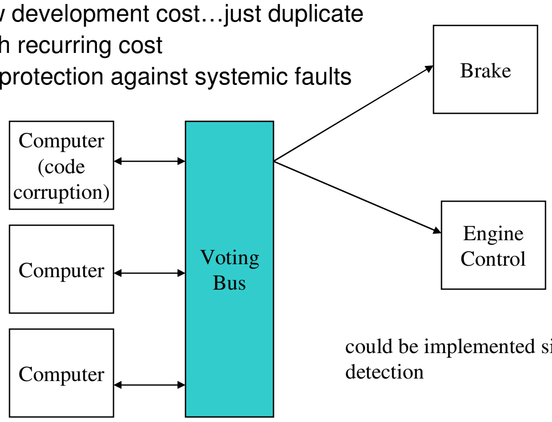
- Self Checking
  - perform periodic checksums on code and data
  - How long does this take?
  - Is Ttest<T0<T1?
  - No protection against systemic faults




- Feasibility of Single Channel Protection
  - Fault Tolerance Time
  - Speed of the processor
  - Amount of ROM/RAM
  - Special Hardware
  - **Recurring cost v. Development cost tradeoff**

## Redundancy

- Homogeneous Redundancy
  - Low development cost…just duplicate
  - High recurring cost
  - No protection against systemic faults



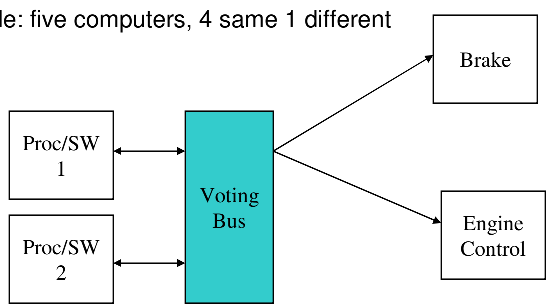

could be implemented similar to collision detection

what happens if you have an even number of computers?



## Diversity

- Heterogeneous Redundancy
  - Protects against random and some systemic faults.
  - Best if implementation teams are kept separated
- Space shuttle: five computers, 4 same 1 different



## Design Process

1. Hazard Identification and Fault Tree Analysis
2. Risk Assessment
3. Define Safety Measures
4. Create Safe Requirements
5. Implement Safety
6. Test,Test,Test,Test,Test



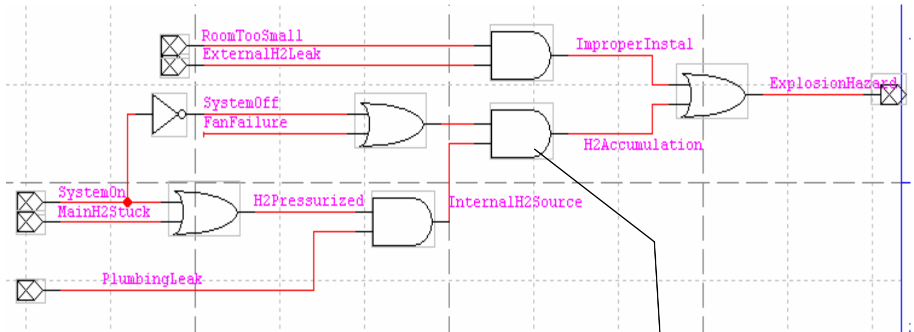
## Hazard Analysis – Working forward from hazards

### Ventilator Example

Human in Loop

| Hazard | Severity | Tolerance Time | Fault Example | Likelihood | Detection Time | Mechanism | Exposure Time |
|---|---|---|---|---|---|---|---|
| Hypo-ventilation | Severe | 5 min. | Motor Too Slow | Rare | 30sec | Indep. pressure sensor w/ alarm | 40sec |
| | | | Esophageal intubation | Medium | 30sec | C02 sensor alarm | 40sec |
| | | | User mis-attaches breathing hoses | never | N/A | Different mechanical fittings for intake and exhaust | N/A |
| Over-pressurization | Sever | 0.05sec | Release valve stuck closed | Rare | 0.01sec | Secondary valve opens | 0.01sec |

## Fault Tree Analysis




Satisfiability Analysis: What combinations of inputs produce the hazard
Explosion Hazard: (SystemOn * FanFailure * PlumbingLeak) +
(SystemOff * MainH2Stuck * PlumbingLeak)
Let Plumbing Leak = 1 (there is always some level of leakage
(SystemOn * FanFailure) + (SystemOff * MainH2Stuck)
Let Tdetect(FanFailure < ToleranceTime)
(MainH2Stuck * System is Off) is our biggest concern.
Mitigation: Open an valve from internal H2 plumbing when off?? Why Not?
**Proper Installation Required!**

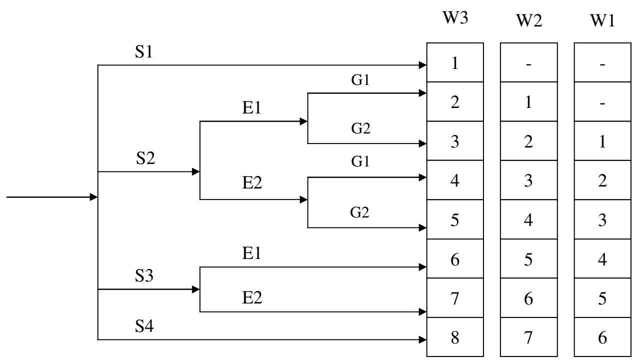


## FMEA: Same as Hazard Analysis, but Start w/ Faults

- Failure Mode: how a device can fail
  - Battery: never voltage spike, only low voltage
  - Valve: Stuck open? Stuck Closed?
  - Motor or Motor Controller: Stuck fast, stuck slow?
  - Hydrogen sensor: Will it be latent or mimic the presence of hydrogen?
- Failure Modes and Effects Analysis
  - Great for single fault tolerant systems
- For each system.
  - Identify all failure modes and likelihoods
  - Identify the hazard that is produced by each failure
  - Determine Time tolerance for each potential hazard
  - Design Considerations
    - Mitigation
    - Detection
  - Response
    - What to do: shutdown, alarm, disable certain features, etc.
- Search space can be quite large

## Risk Assessment

- Risk is orthogonal to hazard analysis
- Determine how risky your system is




S: Extent of Damage
- Slight injury
- Single Death
- Several Deaths
- Catastrophe

E: Exposure Time
- infrequent
- continuous

G: Preventability
- Possible
- Impossible

W: Probability
- low
- medium
- high



## Example Risk Assessment

| Device | Hazard | Extent of Damage | Exposure Time | Hazard Prevention | Probability | TUV Risk Level |
|---|---|---|---|---|---|---|
| Microwave Oven | Irradiation | S2 | E2 | G2 | W3 | 5 |
| Pacemaker | Pacing too slowly<br>Pacing too fast | S2 | E2 | G2 | W3 | 5 |
| Power station burner control | Explosion | S3 | E1 | -- | W3 | 6 |
| Airliner | Crash | S4 | E2 | G2 | W2 | 8 |

## Define the Safety Measures

- Obviation: Make it physically impossible (mechanical hookups, etc).
- Education: Educate users to prevent misuse or dangerous use.
- Alarming: Inform the users/operators or higher level automatic monitors of hazardous conditions
- Interlocks: Take steps to eliminate the hazard when conditions exist (shut off power, fuel supply, explode, etc.
- Restrict Access. High voltage sources should be in compartments that require tools to access, w/ proper labels.
- Labeling
- Consider
  - Tolerance time
  - Supervision of the system: constant, occasional, unattended. Airport People movers have to be design to a much higher level of safety than attended trains even if they both have fully automated control



## Create Safe Requirements: Specifications

- Document the safety functionality
  - eg. The system shall NOT pass more than 10mA through the ECG lead.
  - Typically the use of NOT implies a much more general requirement about functionality…in ALL CASES
- Create Safe Designs
  - Start w/ a safe architecture
  - Keep hazard/risk analysis up to date.
  - Search for common mode failures
  - Assign responsibility for safe design…**hire a safety engineer.**
  - Design systems that check for latent faults
- Use safe design practices…this is very domain specific, we will talk about **software**

## 5. Implement Safety – Safe Software

- Language Features
  - Type and Range Safe Systems
  - Exception Handling
- Re-use, Encapsulation
  - Objects
  - Operating Systems
  - Protocols
- Testing
  - Regression Testing
  - Exception Testing (Fault Seeding)



## What happens if

```
void*  a[SZ];                    // Data Structure Definition
a[i] = (void*) x;                // Range Violation?
x = (myType *)a[i];              // Range and Data Type Violation?
```

__Ideal Error Checking Hierarchy__

Automatic:

Compile Time Checking (Static) better than Run Time Checking (Dynamic)

- data types for assignments
- range
- unitialized
- Out of memory….etc.

Programmer:

Semantic error conditions (e.g array not sorted, too many users, etc)

```
if (i < SZ) a[i] = (void*) x;  else what??          // Range Violation?
if (i < SZ) x =  (myType *) a[i]; else what??       // Range and Data Type Violation?
```

__Four Main Problems in C__

1. Static analysis not defined by the language: a[5] means *(a+5), not "fifth element of the array a".
2. There is no run-time checking. OS checks to make sure you stay in your space.
3. Exception flow is indistinguishable from normal flow and exception handling is voluntary
4. Semantic checking onus on user of data structure

## Language Definition

- **static analysis is up to the compiler**
  - Define the semantics of the language to include all compile time checks that cannot be caught at run time
    - Un-initialized variables
    - type mismatch
- **The run time environment performs dynamic checks** that cannot be caught at compiler time: mainly to make sure that you never access memory the wrong way
  - Null pointer access
  - Array out of bounds
  - Type mismatch even when casting
  - Memory Management and Garbage Collection

```
a[i] = (void*) x;          // raise an exception
x =  (myType *) a[i];      // raise and exception
```

  - What happens in the event of an exception?



## Exception Handling

- Its NOT okay to just let the system crash if some operation fails! You must, at least, get into safe mode.
- In C it is up to the designer to perform error checking on the value returned by f1 and f2. Easily put off, or ignored. Can't distinguish error handling from normal program flow, no guarantee that all errors are handled gracefully.
- **typical C approach:**

```
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
    …
}
```

In C, the exception flow is the same as the normal flow. Use negative numbers for exceptions?!

## Exception Handling in Java

```
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
   }
}
```

Separates
- throwing exceptions
- functional code
- exception handling

All exceptions must be handled or thrown. Exceptions are subclassed so that you can have very general or very specific exception handlers.



## 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);
}
```

## More Language Features

- Garbage collection
  - What is this for
  - Is it good or bad for embedded systems
- Inheritance
  - Means that type safe systems can still have functions that operate on generic objects.
  - Means that we can re-use commonalities between objects.
- Re-use
  - Use trusted systems that have been thoroughly tested
  - OS
  - Networking
  - etc.



## Java for Embedded Systems

- Why not Java for Embedded Systems
  - Its slower
  - Code bloat
  - Garbage Collection may not be interruptible (Latency, predictability)
  - Time resolution – run time support for multithreading and synchronization must be optimized. Java assumes the existence of a basic operating system.
  - Hardware access – interrupt handlers, event handlers
- TinyOS
  - A Component model that seems to be good for “reactive” systems. Probably does a good job of addressing the four major issues listed here.

## Safe Design Process

- Mainly, the hazard/risk/FMEA analysis is a process not an event!
- How you do things is as important as what you do.
- Standards for specification, documentation, design, review, and test
  - ISO9000 defines quality process…one quality level is stable and predictable.