COE718: Embedded Systems Design

Lecture 10:
Fault Tolerant Embedded Systems

Anita Tino

High Performance Emb Sys

- Many safety critical application require:
  - High performance
    - High speed I/O (i.e. From Mb – Gb/sec)
    - Memory
    - Reliable software and hardware
  - Strict performance requirements
    - Deadlines!
    - Reliable

High Performance Emb Sys

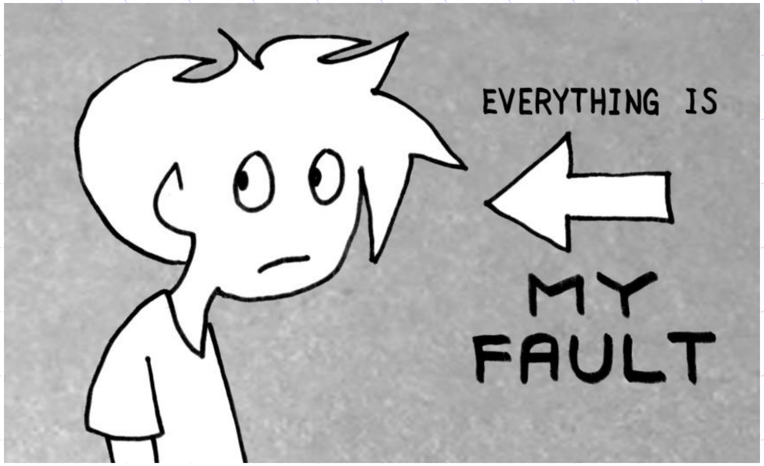
- Many safety critical application require:
  - High performance
    - High speed I/O (i.e. From Mb – Gb/sec)

Fault Tolerance

    - Reliable software and hardware
  - Strict performance requirements
    - Deadlines!
    - Reliable

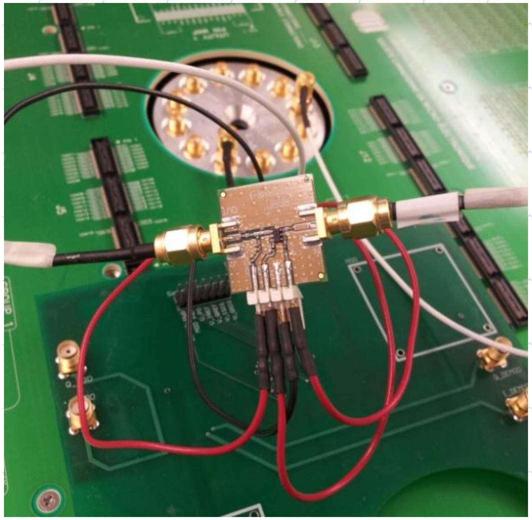
Faults – Why and how?

- **Fault** – erroneous state of software or hardware resulting from component failure(s)



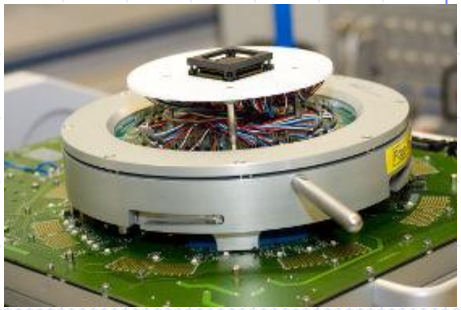
Faults – Why and how?

- **Fault Sources**
  - Design errors
    - Software or hardware
  - Manufacturing Problems
    - Damage, deterioration, stress
  - External disturbances
    - Environment conditions
    - Electromagnetic interference, radiation etc
  - System Misuse



Fault Sources & Types

- Mechanical
  - Wears out, fatigue, overload, corrodes
- Electronic (Hardware)
  - Manufacturing defects
    - Tests
  - Operating Environment – noise, heat, ESD
  - Design defects (i.e. Pentium I)



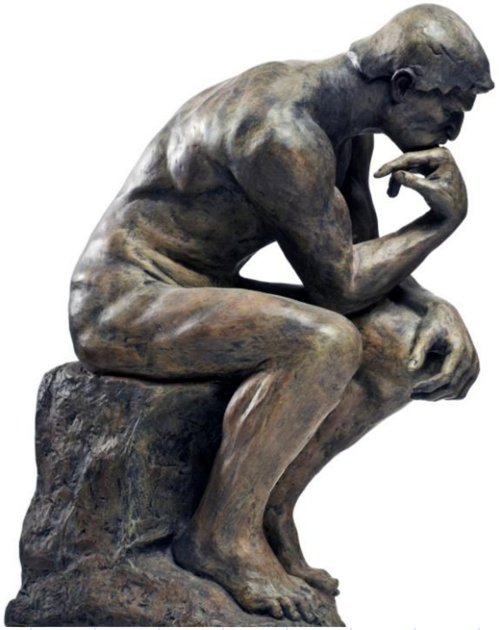
- Software
  - Design defects, run-time faults etc (Mars Pathfinder)
- PEOPLE
  - This lecture is only 3 hours so lets continue…

Fault Type Classifications

- Failure – component does not provide its service
- Fault – defect within the system (result from failure)
- Error – manifestation of a fault
  - System deviates from its required operation

Fault Type Classifications

- Extent – Local or distributed?
  - Independent or dependent?
- Value – Determinate? (stuck) or indeterminate (varying value)?
- Duration –
  - Transient = design error, environment
  - Intermittent = repair by replacement
  - Permanent = repair by (total) replacement



How do we design an embedded system to tolerate faults?

FAULT TOLERANT COMPUTING

Tolerating Faults

- 4 categories on how to deal with system faults and increase system reliability

1. Fault Avoidance

- Prevent the fault from occurring during the design phase
  - Use highly reliable components
  - Conservative design
  - Modular design

Tolerating Faults

2. Fault Tolerance

- Provide a service to comply with the component specification in spite of the fault that occurs
  - Redundancy
  - Fault detection

Fault Tolerance = ability for a system to survive the presence of faults

Tolerating Faults

3. Fault Removal

- Minimize the presence of faults
  - Verify through various tests during different design phases:
    - Initial design, individual components
    - Prototype tests
    - Once manufactured (i.e. Fabbing)
    - Stress tests etc

Tolerating Faults

3. Fault Removal

- Recovering from Faults, 3 classes:

1. Full Recovery
   – May require to switch in another system
2. Degraded Recovery
   – Hopefully graceful degradation
   – Defective component is taken out of service
3. Safe Shutdown
   – Fail-safe operation, 2. may lead to this

Tolerating Faults

4. Fault Forecasting

- How to estimate the presence, occurrence, and consequences of faults
  - Evaluation models, i.e. Reliability, MTTF

- We'll be taking a closer look at points 2. and 4. in class i.e tolerating and forecasting

Fault Tolerance

- Fault detection – gives warning when a fault occurs.
- Basic idea would be *duplication*
  - 2 identical copies of hardware running in parallel, doing same computations
  - Compare results to one another
  - When results don’t match = fault

Redundancy

Hardware

- Use extra components to mask the effect of a faulty component
  - i.e. Masking Redundancy

HW Fault-Tolerant Design Technique

- **Triple Modular Redundancy (TMR)** – 3 identical copies of modules provide separate results to a voter that produces a majority vote

TMR ARCHITECTURES

Redundancy

Software

- Even bigger challenge!
  - Bugs usually show up later
  - Watchdog – but only good if SW crashes
- SW Fault-Tolerant Design Techniques
- **Recovery Block Scheme (RB)**
- **N-Version Programming Scheme (NVP)**

Biggest Software Bugs!

Recovery Block Scheme (RB)

- Employs dynamic redundancy – use an online acceptance test to determine which version to believe
- Scheme consists of:

1. A primary module to execute critical SW functions
2. Acceptance test for output of primary module
3. Alternate modules performing same functions as primary

Recovery Block Scheme (RB)

- Basic algorithm:

Ensure Acceptance Test (AT) met by P

  - P = Primary module

Else use A1

Else use A2

….

Else use An

Else Error

Recovery Block Scheme (RB)

- What are the **acceptance tests**? Can use a variety of distinct approaches:

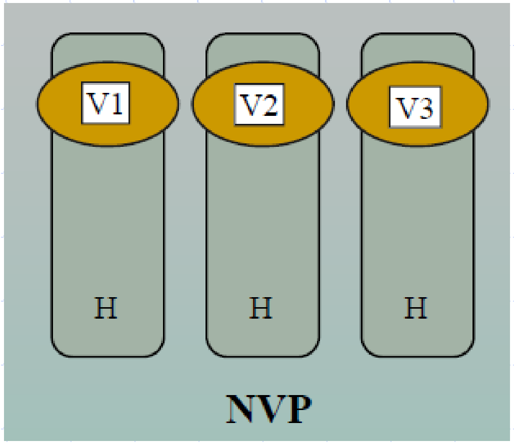
- **Reversal Checks** – take result from module and attempt to calculate what input value was applied, compare and determine if acceptable
  - i.e. If module SQRT(x), AT would pow(x,2) and check input
- **Coding Checks** – verify checksums etc
- **Reasonableness Checks** – Check values before and after execution, see if they've changed, exceptions etc

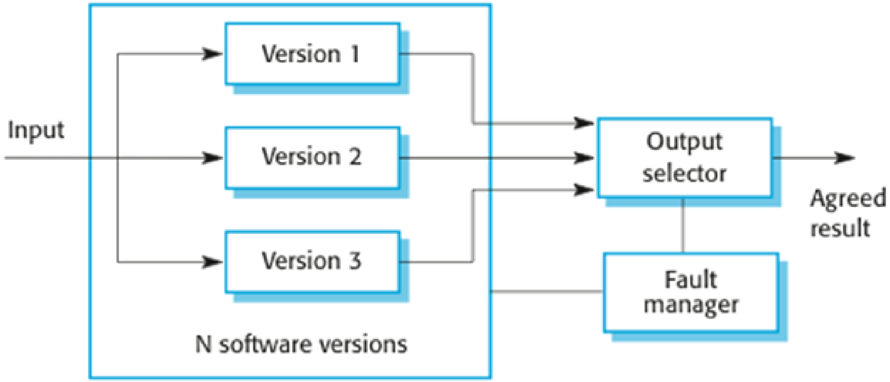
RB ARCHITECTURE

N-Version Programming (NVP)

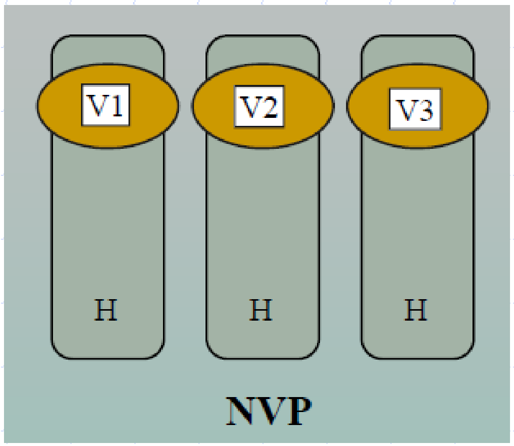
- Create N-independent program variants of the system that execute in parallel
- Each program must be developed using different:
  - Algorithms, or Techniques, or Programming languages etc
- Vote at end – create checkpoints (synchronize programs)
  - Community Error Recovery - Detect errors at the checkpoints, select best of N, or
  - Acceptance test – compare outcome/result

N-Version Programming (NVP)



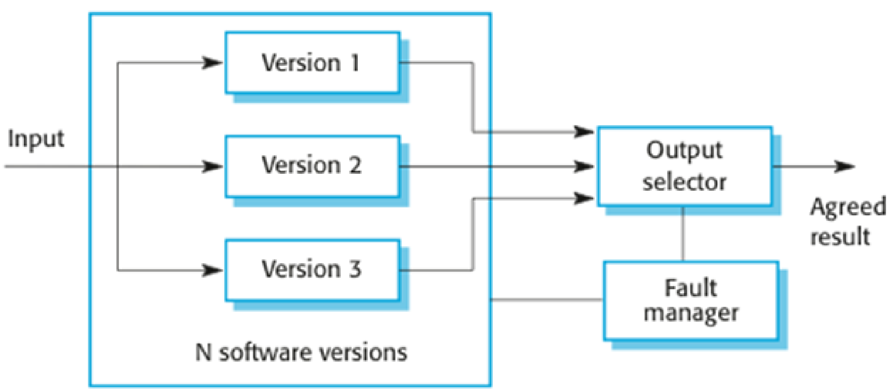


N-Version Programming (NVP)



Unless....

Output Selector

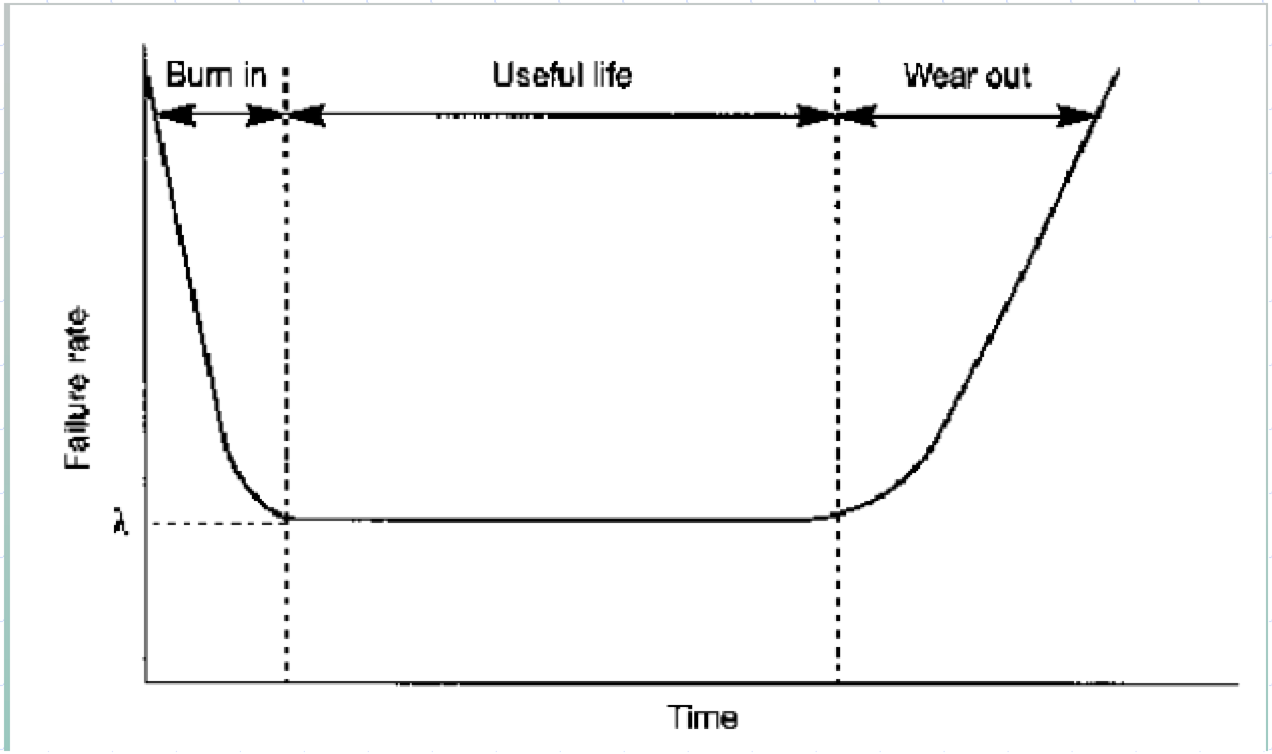


BACK TO FAULT FORECASTING....

System Reliability

- Reliability of a system = $R_f(t)$
  - Probability that no fault of the class F occurs during time t
- $R_f(t) = P[t_{init} \leq t < t_f$, for all f in F]
  - t_{init} is the time system is introduced to service
  - T_f is the time first failure f (of set F) occurred
- $Q_f(t)$ is the failure probability
  - $R_f(t) + Q_f(t) = 1$

System Reliability



- Not so straight forward

-During the system's "useful life", components exhibit a constant failure rate = λ

$$R(t) = e^{-\lambda t}$$

System Reliability

Automotive Embedded System Component	Failure Rate, λ
Military Microprocessor	0.022
Typical Automotive Microprocessor	0.12
Electric Motor Lead/Acid battery	16.9
Oil Pump	37.3

- Failure rates expressed as:

  Failures/million operating hours

Calculating System Reliability

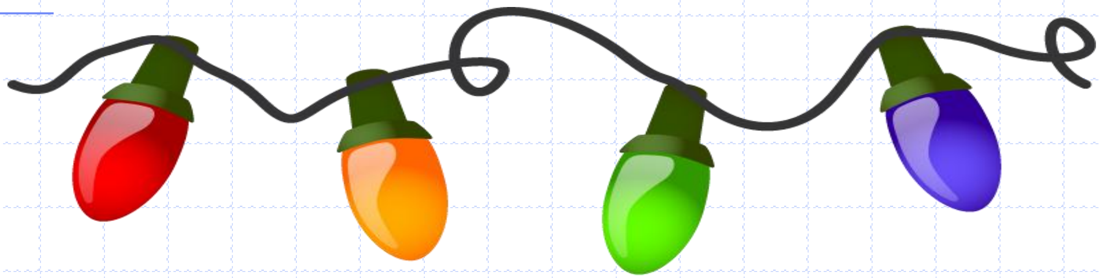
- System reliability depends on the correct functionality of component
- We calculate considering statistical independence
  - Independent if occurrence of one does not affect the probability of the other
  - Encapsulate the components that affect each other

Calculating System Reliability

- We will consider the following 3 connections for system reliability:

1. **Serial** (Connected) System Reliability
2. **Parallel** (Connected) System Reliability
3. **Parallel-Serial** System Reliability

Serial System Reliability



- Overall system reliability depends on the proper working of each component

$$R_{ser}(t) = R_1(t) \times R_2(t) \times R_3(t) \times \ldots. \times R_n(t)$$

$$R_{ser}(t) = \prod_{i=1}^{n} R_i(t)$$

$$\textbf{Serial Failure rate,}\ \lambda_{ser} = \sum_{i=1}^{n} \lambda_i$$

$$R_{ser}(t) = e^{-t\left(\sum_{i=1}^{n} \lambda_i\right)}$$

Parallel System Reliability

- Consider Q(t) when calculating parallel connected components
  - i.e. The failure (Q(t)) of a component is independent of a component that is connect in parallel
- => Q(t) = 1 – R(t)

$$Q_{par}(t) = \prod_{i=1}^{n} Q_i(t)$$

$$R_{par}(t) = 1 - \prod_{i=1}^{n} [1 - R_i(t)]$$

$$Q_k(t) = 1 - e^{-\lambda_k t}$$

SERIAL/PARALLEL CONNECTED SYSTEM EXAMPLES….